Fault Tolerance in Critical Information Systems

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

at the

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Matthew C. Elder

May 2001

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
MAY 2001		**00-00-2001 to 00-00-2001**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Fault Tolerance in Critical Information Systems	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Virginia,Department of Computer Science ,151 Engineer's Way,Charlottesville,VA,22904-4740**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**238**	

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy, Computer Science

Matthew C. Elder

This dissertation has been read and approved by the Examining Committee:

John C. Knight (Advisor)

Alfred C. Weaver (Committee Chair)

Yacov Y. Haimes (Minor Representative)

James P. Cohoon

Anita K. Jones

Accepted for the School of Engineering and Applied Science:

Dean Richard W. Miksad
School of Engineering and Applied Science

May 2001

Abstract

Critical infrastructure applications provide services upon which society depends heavily; such applications require constant, dependable operation in the face of various failures, natural disasters, and other disruptive events that might cause a loss of service. These applications are themselves dependent on distributed information systems for all aspects of their operation, so survivability of these critical information systems is an important issue. Survivability is the ability of a system to continue to provide service, though possibly alternate or degraded, in the face of various types of failure and disruption. A fundamental mechanism by which survivability can be achieved in critical information systems is fault tolerance. Much of the literature on fault-tolerant distributed systems focuses on tolerance of local faults by detecting and masking the effects of those faults. I describe a direction for fault tolerance in the face of non-local faults—faults whose effects have significant non-local impact, sometimes widespread and sometimes catastrophic—where often the effects of these faults cannot be masked using available resources. The goal is to recognize these non-local faults through detection and analysis, then to provide continued service (possibly alternate or degraded) by reconfiguring the system in response to these faults.

A specification-based approach to fault tolerance, called RAPTOR, is presented that enables systematic structuring of formal specifications for error detection and recovery, utilizes a translator to synthesize portions of the implementation from the formal specifications, and provides an implementation architecture supporting fault-tolerance activities. The RAPTOR approach consists of three specifications describing the fault-tolerant system, the errors to be detected, and the actions to take to recover from those errors. The RAPTOR System includes a synthesizer, the Fault Tolerance Translator, to generate implementation of code components from the specifications to perform error detection and recovery activities. In addition, a novel implementation architecture incorporates the generated code as part of an infrastructure supporting fault tolerance at both the node and system levels. Finally, the solution approach is explored and evaluated through the use of case studies and experiments in two critical infrastructure application domains.

Acknowledgments

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0314. This work was also supported under DARPA grant number F30602-99-1-0538. The views and conclusions contained herein are those of the author and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Laboratory, or the U.S. Government.

This material is also based in part on work supported by a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

List of Figures

List of Tables

1

Introduction

The nation is critically dependent upon a number of application domains—among them banking and finance, electric power generation, telecommunications, transportation industries, and military services—for the smooth and proper day-to-day functioning of society. The loss of service in any of these application domains would have serious consequences, thus these domains are referred to as *critical infrastructure applications*.

Many of these application domains, in turn, have become exceedingly dependent upon their information systems to provide their services to society. Recently, much attention has been given to this nation's dependence on these *critical information systems*, focusing on their fragile nature and vulnerable state [58], [61].

Given the dependence upon these critical information systems, an important property that these systems must achieve is survivability. Survivability is one attribute of dependability; other aspects include reliability, availability, and safety [46]. Informally, survivability is the ability of a system to continue to provide service, possibly alternate or degraded, in the face of failures, attacks, or accidents [28]. The failures of concern in critical information systems can include such events as hardware or software failure, operator error, power failure, environmental disaster, or malicious security attacks [61].

One approach to providing the requirement of survivability is fault tolerance. Fault tolerance enables systems to continue to provide service in spite of the presence of faults. Fault tolerance consists of four phases: error detection, damage assessment, state restoration, and continued service [7]. The first two phases constitute comprehensive error detection, and the latter two phases constitute comprehensive error recovery. Survivability is intimately related to and dependent upon both the recognition of certain system faults that affect the provision of service (error detection) and the proper response to these system faults in order to provide some form of continued service (error recovery). The recognition of the system faults most likely to affect the provision of service involves high-level error detection: identifying non-local faults, correlating lower-level faults among application nodes, detecting widespread failures, and recognizing catastrophic events that could significantly disrupt service to the end user. The proper response to these faults—non-local, correlated, widespread, or catastrophic—is often a reconfiguration of application nodes,

sometimes requiring coordination. The focus of this research then is a specific form of fault tolerance, high-level error detection and coordinated error recovery, applied to critical information systems.

In this work, I present a specification-based approach to fault tolerance, called RAPTOR, that enables systematic structuring of formal specifications for error detection and error recovery, utilizes a translator to synthesize portions of the implementation from the formal specifications, and provides an implementation architecture supporting fault-tolerance activities. A RAPTOR specification of fault tolerance consists of three components: a system specification, an error detection specification, and an error recovery specification. System specification uses an object-oriented database to store the descriptions associated with these large, complex systems, while the error detection and error recovery specifications rely on the formal specification notation Z to describe fault-tolerance activities. The RAPTOR System provides a synthesizer, the Fault Tolerance Translator, to generate implementation components for both high-level error detection and coordinated error recovery from the formal specifications. Lastly, the implementation architecture incorporates the generated code at both the node and system levels to achieve fault tolerance. At the node level, a special type of process—called a reconfigurable process—supports application reconfigurations and can be manipulated by generated code components to report low-level failures and to effect recovery responses. At the system level, a supplement to the application network—the Control System—performs high-level error detection and controls error recovery [75], while another architectural supplement—the Coordinated Recovery Layer—coordinates recovery responses across multiple application nodes and processes.

The outline of this document is as follows. The next chapter describes the context of the problem, including relevant characteristics of critical information systems and overviews of two application domains studied in this work. Chapter 3 explores the notion of survivability and outlines a motivating example of survivability from the financial payments domain intended to illustrate the scope of the problem and some requirements for a solution. Chapter 4 presents a framework for the solution—fault tolerance—as well as the characteristics of the faults of interest and the activities that must be undertaken to tolerate those faults. Chapter 5, outlines the solution requirements, discusses a set of principles that guide the solution approach, and presents an overview of the solution. Chapter 6 describes two preliminary systems constructed for problem investigation and the observations and lessons learned from those experiences. The next three chapters (7 through 9) present in detail the aspects of the solution approach: specification of fault tolerance, synthesis of implementation, and the implementation architecture. Chapter 10 discusses the approach to evaluation, a set of key research questions to guide evaluation, and the system for experimentation used in evaluation. Then Chapter 11 describes the set of experiments performed in two critical application domains, as well as analysis of those experiments. Next, chapter 12 addresses the key research questions posed for evaluation. This thesis closes with a survey of related work and conclusions. To supplement the thesis, a set of appendices present the details of the methodology and provide example specifications and systems from the case study application domains.

2

Critical Information Systems

This chapter explores the problem domain in more detail, describing the critical infrastructure applications and outlining the pertinent characteristics of their critical information systems. The balance of the chapter focuses on two particular application domains and their information systems that are the subject of experimentation later in this work.

2.1 Critical Infrastructure Applications

The President's Commission on Critical Infrastructure Protection (PCCIP) cited a variety of infrastructure application domains upon which society has come to rely for normal daily life, the national defense, and economic security [61]. The Commission addressed five different sectors:

- Information and Communications
- Banking and Finance
- Energy (including Electric Power, Oil, and Natural Gas)
- Physical Distribution (including Transportation)
- Vital Human Services (including Water Supply Systems, Emergency Services, and Government Services)

The thrust of the Commission's report was that the infrastructure applications' increasing dependence upon information systems has introduced new threats and vulnerabilities, including "cyber threats" and system interdependencies. To address the wide spectrum of threats, the PCCIP made many recommendations, including a program of infrastructure protection through cooperation and information sharing between industry and government [62].

As appendices to the complete PCCIP report, each of the five sectors produced summary reports. These sector summaries included background information related to each industry, threats and vulnerabilities, findings, and recommendations [61]. (In addition to these sector reports, detailed descriptions of four of these application domains can also be found elsewhere [39]).

A separate study, conducted by the Defense Science Board (DSB) for the Department of Defense, focused on the dependence of national security and military defense systems upon civilian infrastructure systems [58]. The DSB study was concerned with providing an information warfare defensive capability for protection of the infrastructures. This study also cited a great deal of interdependence among the infrastructures, as well as a variety of vulnerabilities in the information infrastructures [58].

Finally, Presidential Decision Directive 63 (PDD-63) instituted a governmental mandate to address the problem of infrastructure protection [18]. The directive required that an initial operating capability be achieved by the year 2000. This has prompted the infrastructure application domains discussed above to institute policies and procedures specific to their industries in response to the government mandate and growing awareness of this problem.

A common theme in all of the studies is that these application domains have become exceedingly dependent upon their information systems for correct and efficient operation. While these application domains have always been carefully protected against certain threats and vulnerabilities, the new dependence upon information systems has opened them up to new sets of concerns and problems, including software failures and malicious security attacks. The next section examines in more detail the information systems upon which these application domains have become critically dependent.

2.2 Critical Information Systems

While the application domains of critical information systems differ greatly, there are common characteristics that are pertinent to the goal of survivable systems. This section explores those characteristics and discusses their significance.

2.2.1 General Characteristics

The architectures of the information systems upon which critical infrastructure applications rely are tailored substantially to the services of the industries which they serve and influenced inevitably by cost-benefit trade-offs. For example, though these systems typically are distributed over wide geographic areas with large numbers of nodes, the application dictates the sites and the distribution of nodes at those sites. Beyond this, however, there are several other similar characteristics possessed by critical information systems in many application domains that are pertinent to achieving the requirement of system survivability:

- *Heterogeneous nodes.* Despite the large number of nodes in many of these systems, a small number of nodes are often far more critical to the functionality of the system than the remainder. This occurs because critical parts of the system's functionality are implemented on just one or a small number of nodes. Heterogeneity extends also to the hardware platforms, operating systems, application software, and even authoritative domains.

- *Composite functionality.* The service supplied to an end user is often attained by composing different functionality at different nodes. Thus, entirely different programs running on different nodes provide different services, and complete service can only be obtained when several subsystems cooperate and operate in some predefined sequence. This is quite unlike more familiar applications such as mail servers routing mail through the Internet.
- *Stylized communication structures.* In many circumstances, critical infrastructure applications use dedicated, point-to-point links rather than fully-interconnected networks. Reasons for this approach include meeting application performance requirements, better security, and no requirement for full connectivity.
- *Performance requirements.* Some critical information systems, such as the financial payment system, have soft real-time constraints and throughput requirements (for checks cleared per second, for example), while others, such as parts of many transportation systems and many energy control systems, have hard real-time constraints. In some systems, performance requirements change with time as load or functionality changes—over a period of hours in financial systems or over a period of days or months in transportation systems, for example.
- *Security requirements.* Survivability is concerned with malicious attacks as well as failures caused by hardware and software faults. Given the importance of critical infrastructure applications, their information systems provide an attractive target to terrorists and other hostile parties intent on disrupting and sabotaging daily life. The deliberate faults exploited by security attacks are of significant concern to a fault-tolerance strategy.
- *Extensive databases.* Infrastructure applications are concerned primarily with data. Many employ several extensive databases with different databases being located at different nodes and with most databases handling very large numbers of transactions.
- *COTS and legacy components.* For reasons of cost and convenience, critical infrastructure applications utilize COTS (Commercial Off The Shelf) components including hardware, operating systems, network protocols, database systems, and applications. In addition, these systems contain legacy components—custom-built software that has evolved with the system over many years.

2.2.2 Future Characteristics

The characteristics listed above are important, and most are likely to remain so in systems of the future. But the rate of introduction of new technology into these systems and the introduction of entirely new types of application is rapid, and these suggest that fault-tolerance techniques must take into account the likely characteristics of future systems as well. I hypothesize that the following will be important architectural aspects of future infrastructure information systems:

- *Larger numbers of nodes.* The number of nodes in infrastructure networks is likely to increase dramatically as enhancements are made in functionality, performance, and user access. The effect of this on fault tolerance is considerable. In particular, it suggests that error detection and recovery will have to be regional in the sense that differ-

ent parts of the network will require different recovery strategies. It also suggests that the implementation effort involved in tolerating faults will increase because there are likely to be many regions and there will be many different anticipated faults, each of which might require different treatment.

- *Extensive, low-level redundancy.* As the cost of hardware continues to drop, more redundancy will be built into low-level components of systems. Examples include mirrored disks and redundant server groups. This will simplify fault tolerance in the case of low-level faults; however, non-local and catastrophic faults will still require sophisticated detection and recovery strategies, analyzing and correlating error information and coordinating recovery and reconfiguration of multiple nodes.
- *Packet-switched networks.* For many reasons, the Internet is becoming the network technology of choice in the construction of new systems, in spite of its inherent drawbacks (e.g., poor security and lack of performance guarantees). However, the transition to packet-switched networks, whether it be the current Internet or virtual-private networks implemented over some incarnation of the Internet, seems inevitable and impacts solution approaches for fault tolerance.

2.3 Example Application Domains

Later in this work, two application domains are the subjects of experimentation and evaluation: (1) banking and the financial payments systems, and (2) electric power generation and distribution. This section provides brief overviews of these two domains and their information systems.

2.3.1 Financial Payments System Description

The nation's payments systems are the core component of the nation's entire financial industry. The payments systems are critical for the efficient functioning of the nation's market economy: the daily business operations of every industry rely on the payments systems to make purchases and sales, pay salaries, and save and invest monetary resources.

There are two primary payments systems, wholesale payments and retail payments. This work will focus on the retail payments system. (For a detailed discussion of payments systems, please refer to a survey of the banking system [39].) Retail payments typically are small-dollar payments used by individuals and businesses as compensation for services rendered or goods supplied; examples include checks, credit card transactions, and Automated Clearing House (ACH) payments.

Checks are the most common form of retail payment. Check clearing is the movement of a check from the depository institution (the bank at which the check is deposited) to the institution on which its funds are drawn, with the funds moving back in the opposite direction, and appropriate credits and debits made to the respective accounts.

In a complex banking system with large numbers of participants, it is inefficient for banks to establish so many bilateral relationships and hold many accounts at correspon-

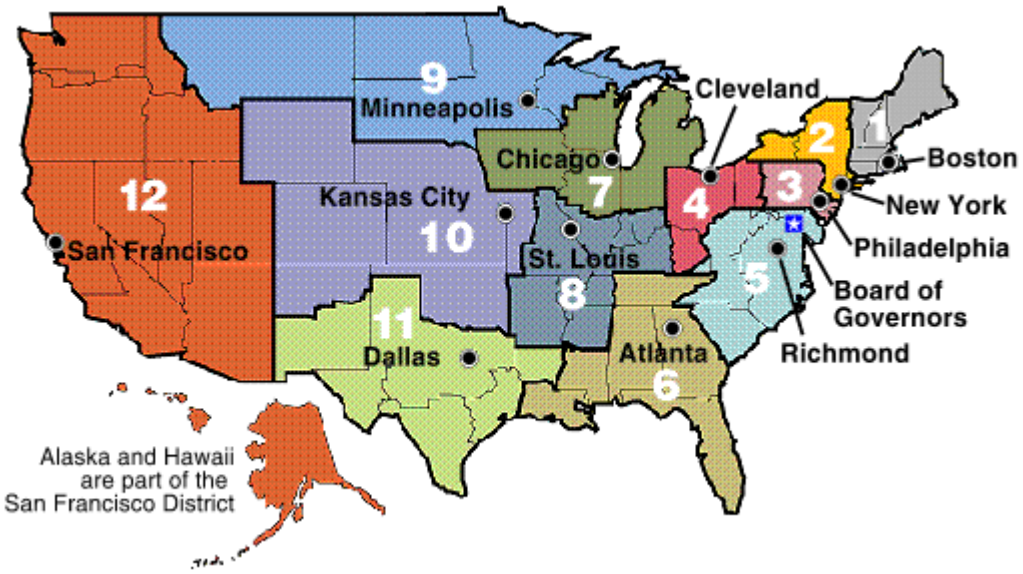

Figure 1: The Twelve Federal Reserve Districts [30]

dent banks for settling purposes. Every bank, however, must be prepared to meet customers' needs to send money to or receive money from any economic entity holding an account at any other bank in the system. This requirement can be satisfied in an efficient manner by a central institution that provides account and settlement services to virtually the entire banking industry. In the United States financial payments system, the central bank is the Federal Reserve [76].

The Federal Reserve System is composed of 12 regional banks along with the Board of Governors in Washington, D.C., 25 branch offices, and 11 regional check processing centers (see Figure 1 [30] and Table 1). Approximately 9,500 of the nation's 26,000 banks are members of the Federal Reserve [39]. Member banks are required to maintain a reserve of ten percent of their total assets with their regional reserve bank. Smaller, nonmember banks have access to the Federal Reserve System through a correspondent relationship with a member bank.

Table 1: Federal Reserve Districts and Branch Offices

Federal Reserve Districts	Additional Branch Offices
Federal Reserve Bank of Boston	(none)
Federal Reserve Bank of New York	Buffalo, NY
Federal Reserve Bank of Philadelphia	(none)
Federal Reserve Bank of Cleveland	Cincinnati, OH Pittsburgh, PA
Federal Reserve Bank of Richmond	Baltimore, MD Charlotte, NC
Federal Reserve Bank of Atlanta	Birmingham, AL Jacksonville, FL Miami, FL Nashville, TN New Orleans, LA
Federal Reserve Bank of Chicago	Detroit, MI
Federal Reserve Bank of St. Louis	Little Rock, AK Louisville, KY Memphis, TN
Federal Reserve Bank of Minneapolis	Helena, MT
Federal Reserve Bank of Kansas City	Denver, CO Oklahoma City, OK Omaha, NE
Federal Reserve Bank of Dallas	El Paso, TX Houston, TX San Antonio, TX
Federal Reserve Bank of San Francisco	Los Angeles, CA Portland, OR Salt Lake City, UT Seattle, WA

Upon deposit of a check for clearance, possibly many banks are required to effect the transfer of funds associated with that check. If the bank at which the check is deposited hosts the account on which the funds will be drawn, then that bank can settle the transaction itself; this is called an *on-us* check. All other checks are *interbank* checks and require routing through the banking system. Typically, the process for interbank checks works as follows:

1. When a bank receives a check to deposit from one of its customers, the check is batched with others for similar destinations (that is if the bank is a Federal Reserve member bank; if it is not, the checks are passed on to the member bank with which the depository bank is affiliated).
2. Batches of checks from these banks are deposited with the Federal Reserve regional bank of which that bank is a member.

3. The Federal Reserve routes batches of checks to the appropriate Federal Reserve regional bank.
4. At the destination regional bank, the checks from the batches are distributed to the appropriate member banks from where the funds will be drawn.
5. When the paying bank receives a check, it debits the appropriate customer account at that bank.
6. Similarly, the Federal Reserve regional bank debits the bank assets for the value of that check and transfers the funds to the original bank.

Information systems at each type of bank are involved at every stage of the check clearing process. At the top level is the Federal Reserve and its information systems. The Federal Reserve consolidated its data processing facilities from sites at each of the twelve regional reserve banks to three data centers and two network operations centers in 1996. The three data centers are located in East Rutherford, NJ, Dallas, TX, and Richmond, VA, and the two network operations centers are in Richmond, VA and Chicago, IL. The Federal Reserve regional banks, not the commercial banks, deal with the processing facilities to effect funds transfers [39].

The information system at the Federal Reserve responsible for payments services is called Fedwire. Fedwire is one of the central applications operated by the Federal Reserve at their primary processing data center, enabling member banks to transfer funds and make wholesale payments [39]. Member banks access Fedwire and other central applications using the Federal Reserve's proprietary software, Fedline Station. The Fedline Station software links directly to Federal Reserve banks over leased lines; services provided include funds transfer, buying and selling of annuities or treasury bonds, and making high-volume, recurring payments via Automated Clearing House [39].

Information systems at commercial banks consist of a variety of applications. For Federal Reserve member banks, there are the systems required to access the Federal Reserve service. For non-member banks, there are networked systems to communicate with the affiliated member bank. In addition to these systems, there are computing centers to manage assets and process transactions, complex database systems to maintain customer accounts, teller machines to provide customer access to services and funds, and so on [80].

Examples from the financial payments systems will be presented throughout this work. In addition, a large-scale model of some of these information systems will be utilized for experimentation later in this work.

2.3.2 Electric Power System Description

As the primary source of power throughout this country, the electric power industry is a key critical infrastructure application domain. Electric power is generated, transmitted, and distributed by a complex system of power companies, utilities, brokers, and merchants. The electric power industry is the last major regulated energy industry in the United States, but currently a movement towards deregulation is greatly altering the manner in which power is sold, distributed, and (to a lesser extent) transmitted [25]. This work, however, will focus on the generation of electric power, a key function of the application domain, and on the reliability of the electric power system.

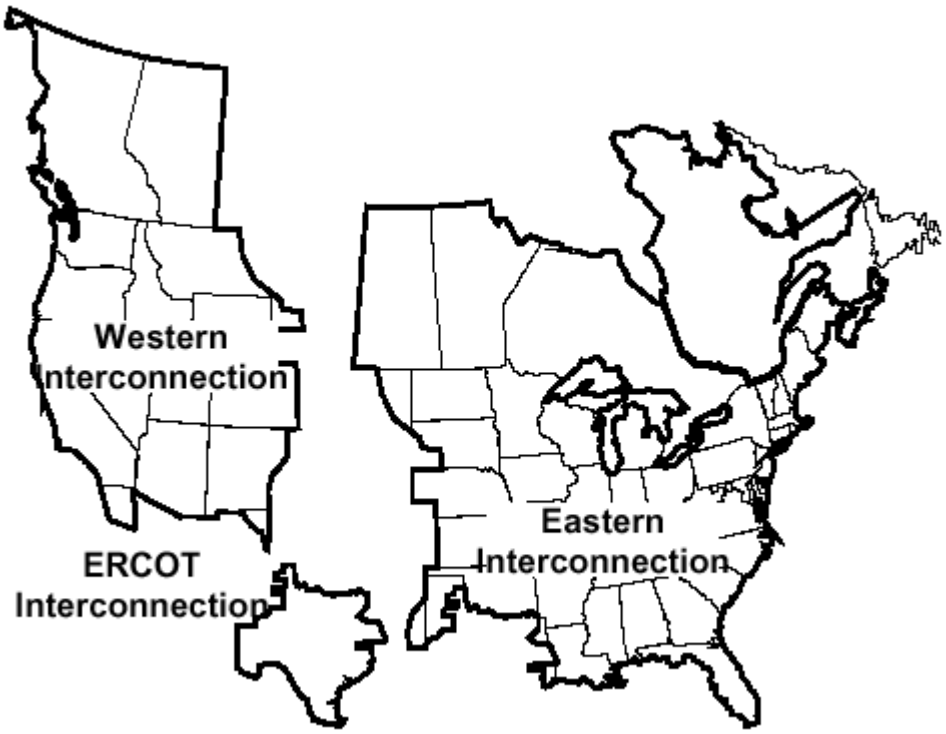

Figure 2: Electric power grid Interconnections [57]

Despite the deregulation of the power industry, the safety and reliability of the electric power system is regulated still by NERC, the North American Electric Reliability Council. The mission of NERC is "to promote the reliability of the electricity supply for North America" [57]. As such, the primary activities of NERC are to regulate the reliable operation of electric power systems in North America, establish policies for operation, and monitor compliance with those policies.

The electric power grid in North America comprises three interconnections—the Eastern Interconnection, Western Interconnection, and ERCOT Interconnection—as pictured in Figure 2 [57]. Interconnections are major networks of electrical power systems; only limited direct-current connections exist between the Interconnections. The relative isolation of the Interconnections limits the trading of power from one region of the country to another, but also ensures that power failures in one portion of the country cannot cascade outside of that particular Interconnection.

NERC consists of ten regional councils, or control regions, throughout the United States, Canada, and Baja California Norte (Mexico) [57]. The members of the regional councils represent all segments of the electric industry, including investor-owned utilities, independent power producers, independent power marketers, electric cooperatives, and various government entities. Regional councils coordinate bulk power policies regarding reliability and service in their region [39]. The ten control regions are presented in Table 2.

Each control region contains some number of control areas (as shown also in Table 2).

Table 2: NERC Control Regions

Control Regions	Interconnection	Number of Control Areas
East Central Area Reliability Coordination Agreement (ECAR)	Eastern	17
Electric Reliability Council Of Texas (ERCOT)	ERCOT	9
Florida Reliability Coordinating Council (FRCC)	Eastern	12
Mid-Atlantic Area Council (MAAC)	Eastern	1
Mid-America Interconnected Network (MAIN)	Eastern	15
Mid-Continent Area Power Pool (MAPP)	Eastern	15
Northeast Power Coordinating Council (NPCC)	Eastern	5
Southeastern Electric Reliability Council	Eastern	22
Southwest Power Pool	Eastern	17
Western System Coordinating Council	Western	30

Formally, a control area is “an electrical system bounded by tie-line metering and telemetry” [57]. What this means is that each control area can monitor and control its power generation and alternating-current frequency in order to provide dependable electric power to customers. In addition, control areas balance actual and scheduled interchange of power with other control areas.

The electric power system is dependent on various information system elements to ensure the proper functioning and reliability of the power grid. At the lowest level, SCADA (Supervisory Control And Data Acquisition) systems provide local level control of power distribution, such as substations, feeder devices, and other distribution elements [39]. Some of the functions provided by SCADA systems include remote supervisory control, data acquisition, alarm and event processing, energy accounting, and data management. SCADA systems report both normal activity and anomalous behavior to power company information systems.

At the level of power companies and control area information systems, EMS (Energy Management Systems) and DMS (Distribution Management Systems) programs provide many different functions related to power generation, transmission, and distribution [39]. Some examples functions performed by EMS programs include power generation and control, transmission control, network analysis, and contingency planning. DMS programs typically provide display of real-time power network status, control of circuit breakers, and real-time power flow calculations [39].

A model of these information systems will be explored later in this work.

3

Survivability

This chapter elaborates on the definition of survivability, then presents a motivating example to help illustrate the issues involved in the provision of survivability for critical information systems.

3.1 Definition of Survivability

Survivability with respect to information systems is a relatively new research area. As such, the precise definition of survivability is still being debated, with a number of definitions proposed [26], [28], [41]. An informal definition of survivability is "the ability [of a system] to continue to provide service (possibly degraded or different) in a given operating environment when various events cause major damage to the system or its operating environment" [41]. This informal definition suggests a number of key points regarding the notion of survivability:

- Survivability is a system property, relating the level of service provided to the level of damage present in the system and operating environment.
- A system must be capable of providing different levels of service. In a system free of damage, the level of service should equate to full functionality. Different levels of service will correspond to varying subsets of functionality, where some functions that a system performs are obviously more critical than others [41].
- The events that cause major damage can range from failures to attacks to accidents. It is often difficult to determine immediately the cause of damage, e.g. whether damage is the result of an intentional security attack or random failures [28]. More important is the effect of the event in terms of damage to the system and operating environment—the amount of damage is central to the level of service that a survivable system can and should provide.

Intuitively then, the notion of survivability involves a system providing full functionality when no damage is present and different subsets of "critical" functionality when the system has been damaged, depending on the type and extent of the damage [41]. To move

beyond an intuitive notion of survivability to a precise definition would require a formal specification of service levels, a formal specification of damage circumstances, and a function relating the two specifications. These specifications describe a set of *survivability requirements*: what must be achieved and not necessarily how a system would go about achieving those requirements. These formal requirements constitute a precise *survivability specification* [41].

It is not the goal of this research, however, to propose another definition of survivability. It should be sufficient to have an informal but clear understanding of survivability requirements for a particular system in order to address the provision of survivability.

3.2 Motivating Example

To illustrate better the problem of achieving survivability in a critical information system, this section presents a motivating example using a hypothetical financial payments system, first describing the system and then exploring its survivability requirements.

3.2.1 System Description

The United States financial payments system, upon which this survivability example is based, is an exceedingly complex system of systems providing a wide array of services to end users. Value transfer, the application explored in this example, is only one of those services (albeit a key one). The system described in this section and the service provided by this example system are highly simplified for the purposes of illustration. For a detailed treatment of the U.S. financial payments system please refer to the text by Summers [76].

The architecture of the information system serving the payments system is roughly a hierarchic, tree-like network as depicted in Figure 3. This model consists of four levels of banks:

- At the top level of the system is the main processing center of the Federal Reserve, and this model includes a single backup.
- The second level of the network consists of the twelve Federal Reserve regional banks, which serve commercial bank customers in their respective regions.
- At the third level of the system are the approximately 9,500 commercial banks that are members of the Federal Reserve; these institutions (typically, large banks) maintain accounts with the Federal Reserve and can transfer value with other banks that have accounts at the Federal Reserve.
- At the lowest layer are the remaining commercial banks that are not members of the Federal Reserve but must be affiliated with member banks in order to effect value transfer [39].

Processing a retail payment (e.g., an individual check) in this system proceeds roughly as follows. At the lowest level, nodes accept checks for deposit, create an electronic description of the relevant information, and forward the details to the next level in the hierarchy. At the next level, payments from different banks are collected together in a

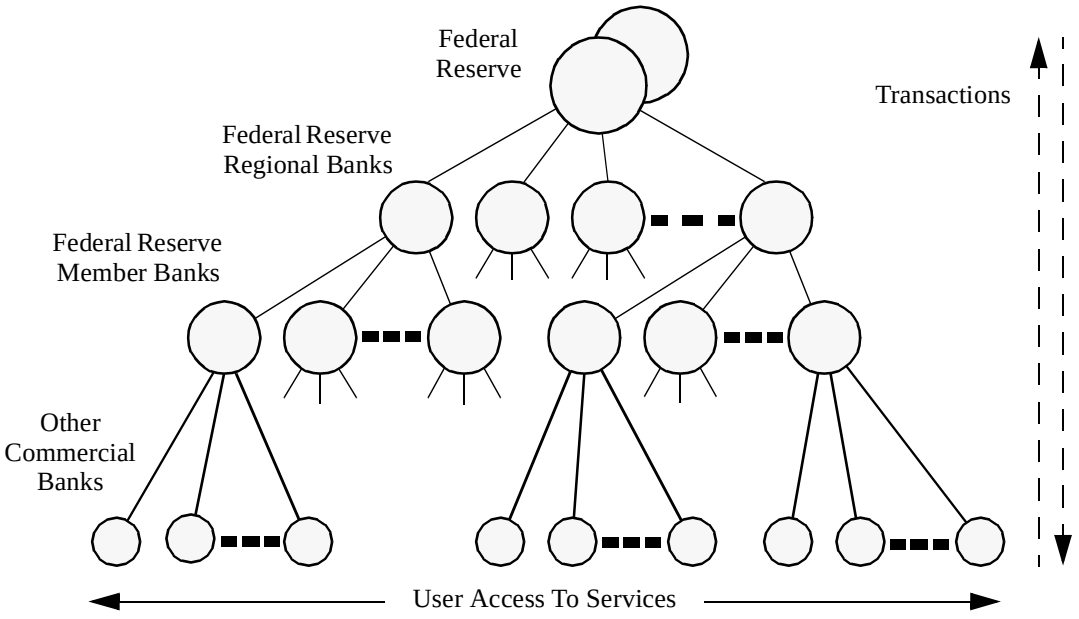

Figure 3: Example payments system architecture

batch and the details forwarded to the Federal Reserve system. Periodically throughout the day the Federal Reserve transfers funds between accounts that it maintains for commercial banks, thus making provision for funds to service the payments requested in the individual checks. The funds are then disbursed through the system to the individual user accounts. Large commercial payments can also originate electronically and are handled individually as they arrive [38].

Again, this system is a gross simplification of the actual U.S. financial payments system. For example, the Federal Reserve's central processing site is in reality a set of geographically separate facilities that serve as backup sites for one another. In addition, there are commercial clearing houses that perform the same value transfer services as the Federal Reserve; in certain circumstances commercial banks will use these clearing houses for some of their check clearing transactions [39].

3.2.2 Survivability Requirements

Given this hypothetical description of a critical information system, this section now considers what must be done in order to achieve survivability.

Again, the informal definition of survivability is "the ability [of a system] to continue to provide service (possibly degraded or different) in a given operating environment when various events cause major damage to the system or its operating environment" [41]. It follows then that in order to discuss the survivability requirements of a system, one must be able to describe the following information:

- The levels of service, including alternate and degraded service modes, that are pro-

vided by the system

- The operating environments of concern, i.e., the state of the system and its operating environment after various events cause damage
- A mapping from the operating environment to the desired level of service for that environment

Firstly then, one must consider and document all of the events and circumstances that

Table 3: Survivability requirements summary

Damage	Level of Service
Multiple local banks fail (wide-area power failure, wide-area environmental stress, common-mode software failure).	*On failure:* Local banks cease service. Regional center starts minimal user services (e.g., electronic funds transfer for selected major customers only) and buffers transactions for local banks. *On repair:* Local banks inform regional center as they are repaired and resume normal service. Regional center transmits transaction backlog, resumes transaction transmission, and terminates minimal user services.
Security penetrations of multiple local banks associated with a single commercial bank (coordinated security attack).	*On failure:* Each local bank ceases service and disconnects itself from the network when its local intrusion alarm is raised. Federal Reserve suspends operations with commercial member bank under attack and buffers all transactions for that bank. *On repair:* Reset all nodes owned by commercial bank under attack. All nodes change cryptographic keys and switch to aggressive intrusion detection. System-wide restart of crucial services temporarily, then resume full service level.
Regional center primary site fails (power failure, hardware failure, software failure, operational error, environmental stress).	*On failure:* Primary site ceases service. Backup site starts service. All connected nodes informed. All communications—up to Federal system and down to branches—switched from primary to backup. Appropriate service selection made—full or reduced. Appropriate switch selection made—instantaneous or delayed. *On repair:* Primary site informs all connected nodes that it is repaired. Primary site databases synchronized. Communications switched. Primary site resumes services in prescribed priority order. Backup site ceases service.
Regional center primary *and* backup sites fail (wide-area power failure, common-mode software failure, wide-area environmental stress, terrorism).	*On failure:* Primary and backup sites cease service. All connected nodes informed. Previously identified local bank processing center designated as temporary replacement for regional facilities. All services terminated at replacement facility, minimal regional service started at replacement facility (e.g., account services for commercial and government clients only). *On repair:* Regional service restarted by resuming applications in sequence and resuming service to local banks in sequence within an application. Minimal service on replacement facility terminated.

could lead to a major loss or disruption of service. In practice, system engineers and domain experts would conduct hazard analysis to determine the system vulnerabilities and threats of concern. The probabilities of the various events could be determined and then requirements analysis could be conducted.

Secondly, a complete survivability specification must document precisely the prescribed level of system service for all of those circumstances of damage that the system is required to handle. Hypothetical examples of the possible damage scenarios and their high-level service responses for the simplified version of the payments system are shown in Table 3. Included in the table are events ranging from the loss of multiple leaf nodes (branch banks) to the loss of a critical node and its backup facilities. In Table 3, the first column describes the damage and parenthetically gives possible causes for the event. In the second column, prescribed levels of service for the damage circumstance are described, including services during the damage event and after its repair.

Note that many low-level failures might occur that are not relevant to the survivability specification because they do not affect the survivability of the system, in that they do not cause major disruptions in the provision of service. For example, the failure of a single commercial branch bank is not described in the survivability specification: while the failure of a local bank might be an inconvenience to customers of that particular bank, that occurrence will not disrupt service to a significant portion of the nation and thus does not impact the survivability of the system. (However, the occurrence of low-level failures must be noted in order to detect widespread events and analyze for correlated events, such as coordinated security attacks or common-mode software failures.)

For purposes of illustration, one particular damage scenario is examined in more detail to explore further the survivability requirements in this system. The event used for illustration is the complete loss of the top-level node of the financial payment system—the Federal Reserve system's main data center and its backup facilities. Using the highly simplified architecture of the payment system in this example, it is assumed that this node consists of a single processing entity with a single backup that maintains mirror image databases. The actual Federal Reserve system utilizes a much more sophisticated backup strategy, of course. The survivability requirements for this damage scenario are the following:

- *Damage:* Federal Reserve main processing center and backup failures (common-mode software failure, propagation of corrupt data, terrorism).
- *On failure:* Complete suspension of all payment services. Entire financial network informed (member banks, other financial organizations, foreign banks, government agencies). Previously identified Federal Reserve regional bank designated as temporary replacement for Federal Reserve facilities. All services terminated at replacement facility, minimal payment service started at replacement facility (e.g., payment service for federal agencies only). All major client nodes redirect communication.
- *On repair:* Payment system restarted by resuming applications in sequence and resuming service to member banks in sequence within each application. Minimal service on replacement facility terminated.

For this particular event, it is assumed that all processing ceases immediately. This is actually the most benign damage circumstance that the system could experience at the top-

level node. More serious events that could occur include undetected hardware failures in which data was lost, a software failure that corrupted primary and backup databases, or an operational failure in which primary data was lost.

The details of the survivability scenario described in this example are possible from the computer science perspective, as are many others. What the banking community requires in practice depends upon the many details and priorities that exist within that domain and are probably far more elaborate than in this example. However, this example does illustrate some of the issues that have to be considered in the provision of survivability.

An important aspect of survivability that is omitted from this example is the need to cope with multiple sequential damage events. It will be the case in many circumstances that a situation gets worse over time; the effect to the end user will be continued degradation of the service that can be provided by the failing system. For example, a terrorist attack on the physical equipment of a critical information system might proceed in a series of stages. The attack might be detected initially during an early stage (although it is highly unlikely that the detection process would be able to diagnose a cause in this case) and the system would then take appropriate action. Subsequent failures of physical equipment would have to be dealt with by a system that had already been reconfigured to deal with the initial attack. This complicates the provision of survivability immensely.

4

Solution Framework: Fault Tolerance

This chapter proposes the solution framework of fault tolerance for the provision of survivability in critical information systems. Fault tolerance is justified as the mechanism for achieving survivability requirements, then the special class of faults that this research targets is described and the impact this has on the fault-tolerance activities of error detection and error recovery is explored.

4.1 Solution Framework

The overarching goal in this work is to achieve survivability in critical information systems. It is important to note that there are often many mechanisms available for accomplishing the same goal. From the literature on dependability (survivability being one of the dimensions of dependability), there are four means for providing dependable computing systems: fault avoidance, fault tolerance, error removal, and error forecasting [14], [46]:

- *Fault avoidance* prevents the occurrence of faults by construction.
- *Fault tolerance* provides service complying with the specified function in spite of the occurrence of faults.
- *Fault removal* minimizes the presence of faults by verification.
- *Fault forecasting* estimates the presence, creation, and consequences of faults by evaluation.

This research explores the provision of survivability in critical information systems using *fault tolerance* for a number of reasons. Firstly, these systems are too complex, too large, and too heterogeneous for it to be possible to prevent the occurrence of all faults during construction using fault avoidance; inevitably there will be faults present in these systems (as in any complex software system). Similarly, fault removal alone cannot obviate the presence of all faults; systems such as these are too complex and large to be verified completely and correctly. Fault forecasting would be difficult for much the same reason. Finally, one of the major threats to survivability in these information systems is malicious security attacks. It is practically impossible to anticipate and prevent all of the

deliberate faults exploited by malicious security attacks, and therefore these faults cannot be avoided: they must be tolerated, if at all possible [5]. Fault tolerance is the most promising mechanism for achieving survivability requirements.

Before proceeding with a discussion of fault tolerance, it is important to understand the precise definitions of the terms fault, error, and failure [46]:

- A *failure* refers to an occurrence of the delivered service deviating from the specified or expected service.
- An *error* is the cause of a failure: an error refers to the erroneous part of the system state that leads to the failure.
- A *fault* is the cause of an error: the occurrence of a fault in the system manifests itself as an error.

Given these definitions and fault tolerance as a solution framework, Anderson and Lee identified four phases to fault tolerance: error detection, damage assessment, state restoration (error recovery), and continued service [7]:

- *Error detection* determines the presence of a fault by detecting an erroneous state in a component of the system.
- *Damage assessment* determines the extent of any damage to the system state caused by the component failure and confines that damage to the extent possible.
- *State restoration* achieves recovery from the error by restoring the system to a well-defined and error-free state.
- *Continued service* for the system, in spite of the fault that has been identified, means that either the fault must be repaired or the system must operate in some configuration where the effects of the fault no longer lead to an erroneous state.

As mentioned previously, the first two phases of fault tolerance constitute comprehensive error detection and the latter two phases constitute error recovery. Given fault tolerance as a solution framework, it is important to consider in more detail the types and characteristics of the faults with which this research is concerned, as well as the impact this will have on the fault-tolerance activities of error detection and error recovery.

4.2 Faults and Fault Tolerance

Because the goal of this work is to enhance survivability, this research is concerned with the need to tolerate faults that directly compromise the survivability of a critical information system. In general, these types of faults are those that affect significant fractions of a network application, faults referred to as *non-local*. Thus, for example, a widespread power failure in which many application nodes are forced to terminate operation is a non-local fault. The complete failure of a single node upon which many other nodes depend would also have a significant non-local effect and could be considered a non-local fault.

Non-local faults have the important characteristic that they are usually *non-maskable*—that is, their effects are so extensive that normal system service cannot be continued with the resources that remain, even if the system includes extensive redundancy [9]. This work is not concerned with faults at the level of a single hardware or software component, referred to as *local* faults. It is assumed that whenever possible local

faults are dealt with by some mechanism that masks their effects. Thus synchronized, replicated hardware components are assumed so that losses of single processors, storage devices, communications links, and so on are masked by hardware redundancy [14]. If necessary, more sophisticated techniques such as virtual synchrony can be used to ensure that the application is unaffected by local failures [10].

Non-local faults might affect a related subset of nodes in a network application leading to the idea that they can be *regional*. Thus, a fault affecting all the nodes in the banking system in a given city or state would be regional, and an appropriate response to such a fault might depend on the specific region that was affected. Similarly, a fault affecting all the nodes within a particular administrative domain (independent of geographic location) would be considered regional. It is also possible that the elements of a non-local fault would manifest themselves over a period of time rather than instantly. This leads to the notion of *cascading* or *sequential* faults in which application components fail in some sequence over a possibly protracted period of time. Detecting and diagnosing such a situation correctly is a significant challenge. Handling these faults is also complicated because the proper response will depend on the system state at the time of the fault.

As mentioned previously, tolerating a fault requires first that the effects of the fault be detected—*error detection*—and second that the effects of the fault be dealt with—*error recovery* [37]. Both error detection and error recovery have to be defined precisely if a fault-tolerant system is to be built and operated correctly, and several issues arise in dealing with these activities.

4.2.1 Error Detection

Error detection for a non-local fault requires the collection of information about the state of the application and analysis of that information. Analysis is required to permit a conclusion about the underlying fault to be made given a spectrum of specific information.

A key problem dealing with error detection in large distributed systems is defining precisely what circumstances are of interest. Events will occur on a regular basis that are associated with faults that are either masked or of no interest. These events have to be filtered and incorporated accurately in the detection of errors of interest. The possibility of false positives, false negatives, and erroneous diagnosis is considerable. In a banking system, for example, it is likely that local power failures are masked routinely, yet if a series of local failures occurs in a specific sequence, they could be part of a widespread, cascading failure that needs to be addressed either regionally or nationally.

4.2.2 Error Recovery

Error recovery for non-local faults requires that the application be reconfigured following error detection. The goal of reconfiguration is to effect changes such as terminating, modifying, or moving certain running applications, and starting new applications. In a banking application, for example, it might be necessary to terminate low priority services, such as on-line customer enquiry, and modify other services, such as limiting electronic funds

transfers to corporate customers.

Unless provision for reconfiguration is made in the design of the application, reconfiguration will be ad hoc at best and impossible at worst [28]. The provision for reconfiguration in the application design has to be quite extensive in practice for three reasons:

- The number of fault types is likely to be large and each might require different actions following error detection.
- It might be necessary to complete reconfiguration in bounded time so as to ensure that the replacement service is available in a timely manner.
- Reconfiguration of multiple application nodes might require coordination to ensure consistent application behavior.
- Reconfiguration itself must not introduce new security vulnerabilities.

Just what is required to permit application reconfiguration depends, in large measure, on the design of the application itself. Provision must be made in the application design to permit the service termination, initiation, and modification that is required by the specified fault-tolerant behavior.

4.3 Solution Strategy

This research focuses on a special class of faults—non-local—that requires a specialized form of fault tolerance. Because non-local faults affect a large subset of application nodes, this impacts the activities of error detection and error recovery and necessitates novel solution strategies for both.

Error detection for non-local faults requires extensive support for analysis of application state in order to recognize regional, cascading, and sequential faults. While it is assumed that local faults can be detected and handled, detection of non-local faults involves collection of low-level error detection data and critical examination of that data. The strategy for high-level error detection must provide for these capabilities.

Error recovery for non-local faults requires a non-masking approach to recovery: application reconfiguration. In traditional fault-tolerant systems, faults typically are masked: following restoration of the system state, continued service is the same as prior to the fault [32]. (Please see the chapter on Related Work for a detailed presentation and discussion.) Masking, however, depends on there being sufficient redundancy in the system to continue to provide the same operation despite the effects of faults. In any system, there are inevitably going to exist faults for which the system has not provided enough redundancy to mask them [14]. In order to tolerate faults in a non-masking manner, an application must reconfigure the remaining resources to provide continued service that is different from that provided prior to the fault. Application reconfiguration provides the mechanism by which alternate service can be provided by the system, and thus survivability can be achieved in those circumstances where the faults could not be masked.

5

Overview of the Solution Approach

Given the solution framework of fault tolerance, the focus on non-local faults, and the high-level strategy presented in the previous chapter, this chapter examines the requirements for a solution, explores principles that will dictate the shape of the solution, and presents an overview of the solution approach.

5.1 Solution Requirements

The characteristics of critical information systems described in Chapter 2 define the solution requirements used to evaluate this research:

- *Scale*. Perhaps the most important and defining element of these critical information systems that must be accommodated is scale. Current information systems are already large, wide-area, distributed systems, and the scale of future systems is likely to increase further. The shear size in terms of number of nodes and links, hardware and software elements, and different modes of functionality drives many elements of a solution approach. The fault-tolerance solution should accommodate systems on the order of thousands of computing nodes, as well as their relevant faults and responses.
- *Heterogeneity.* Another key characteristic that has great bearing on the solution approach is the heterogeneity present in critical information systems. The wide range of platforms, modes of functionality, services provided, and component criticality makes the task of understanding and describing the system much more difficult. In addition, the faults and responses specific to each type of node must be accommodated by the fault-tolerance methodology.
- *Complexity.* The complexity of critical information systems derives in part from the scale and heterogeneity of these systems, but it is also inherent in the architecture and functionality of the systems themselves. The system complexity translates into complexity in the fault-tolerance activities of error detection and error recovery. Reconfiguration of a complex system relies on precise description of that system, the faults of interest, and the actions that must be performed in response to faults.

- *Performance*. Given that most of these applications have performance requirements, the fault-tolerance solution must respect and achieve those goals as much as possible given the circumstances of the failure. This means in general that fault tolerance will have to be achieved within time bounds dictated by the application. In addition, performance-related aspects of errors, such as the timing of faults and detection time, must be accommodated.

It is important to note that this research does not address specific issues arising from COTS and legacy software, complex databases, and security concerns. The solution approach should not preclude any efforts to handle these concerns, but these dimensions of the solution requirements are not the focus of this work. In addition, while support for varying communications structures (both circuit-switched and packet-switched networks) is important, the solution approach should operate at a higher-level of abstraction, independent of underlying communications mechanisms.

5.2 Solution Principles

The pertinent characteristics of the application domains, the solution requirements, and the high-level solution strategy of application reconfiguration to tolerate non-local, catastrophic faults suggest three solution principles to guide this research:

- The use of formal specification
- Synthesis of implementation
- An implementation architecture enabling fault-tolerance activities

These solution principles are explored in the following subsections.

5.2.1 The Use of Formal Specification

The first principle of a solution approach for building fault-tolerant critical information systems is the use of formal specification. The use of formal specification derives from the need to address the solution requirements presented previously. The size of current and expected critical information systems, the variety and sophistication of the services they provide, and the complexity of the survivability requirements mean that an approach to fault tolerance that depends upon traditional development techniques is infeasible in all but the simplest cases. The likelihood is that future systems will involve tens of thousands of nodes, have to tolerate dozens, perhaps hundreds, of different types of fault, and have to support applications that provide very elaborate user services. Programming error detection and recovery in such systems using conventional methods is quite impractical.

There are many advantages to working with specifications as opposed to implementations:

- First and foremost is the ability to specify solutions at a high level, thereby abstracting away to some extent the details of working with so many nodes, of so many different types, that provide so many different services. An implementation-based solution would require an inordinate amount of effort, dealing with such a wide variety of nodes, applications, errors, and recovery strategies at a lower level of detail.

- Secondly, specifications provide the ability to reason about and analyze solutions at a higher level of abstraction [82]. Depending on the notation and degree of formality, various forms of syntactic and semantic analysis are possible with specifications, providing the opportunity to ensure various dimensions of correctness before implementation [27].
- Finally, if an implementation can be synthesized from a specification, this would allow fault-tolerance strategies to be investigated and changed relatively quickly: different fault-tolerance schemes and designs can be rapidly prototyped and explored using a specification-based approach.

As discussed in Chapter 4, to make a critical information system fault tolerant, it is necessary to introduce mechanisms to recognize the errors of interest, maintain state information about the system to the extent that it affects error detection and error recovery, and define the required error recovery responses from all relevant system states. Each of these activities should be specified clearly and correctly to facilitate design and implementation. An important activity early in this research was precise definition of key specification elements, followed by determination of appropriate notations for each of these sub-specifications, as well as integration of these specification components. The appropriate formal notations enable the requirements to be expressed naturally and facilitate various forms of syntactic and semantic analysis.

5.2.2 Synthesis of Implementation

The second desirable principle of a solution approach is that as much of the implementation as possible be generated from the specification. Again, synthesis of implementation derives from the need to address the solution requirements, as well as being related to the use of formal specification. There are a number of advantages to being able to synthesize implementation from formal specification:

- Firstly, synthesis allows the system builder to leverage off of the formalism of the specification notations, thus amortizing the cost and effort involved in the development of a formal specification [82].
- In addition, in terms of validation and verification effort, time and resources can be spent on validating and verifying the specification and the synthesis mechanism(s) rather than every version of the system implementation (where the cost of change further along the software life cycle is more expensive).
- Finally, as mentioned previously, synthesis of implementations from formal specification enables rapid prototyping of alternatives in fault-tolerance strategies and designs. Rapid prototyping of synthesized implementations from formal specifications—without incurring much of the cost of development from scratch for every fault-tolerance design—facilitates a risk-driven, spiral-model development process.

While as much of the implementation as possible should be generated from the formal specifications, there will be limitations on how much code can be generated, of course. Specification typically is concerned more with “what” needs to be done than “how” something is to be done, thus there is inevitably a gap that must be resolved. Given that this

research will not be solving the general code generation problem, it will not be possible to generate the entire implementation from a high-level, formal specification.

Synthesis involves a trade-off as well: the construction of a translator to generate implementation components has the benefits mentioned above, but it forces some lower-level design decisions to be resolved in order to fix the target implementation structure. Given that the target for this translator is not general-purpose code though—the target is a set of fault tolerance-specific implementation components—this design trade-off is not necessarily a major issue.

The implementation components to be generated are code to help perform fault tolerance. In terms of error detection, the various sequences of events that constitute high-level errors are being specified, so the translator should be able to generate code to effect recognition of those occurrences. In terms of error recovery, the specification outlines the particular activities an application must perform upon occurrence of system errors, so the translator should be able to generate code to prompt the application to respond or reconfigure under the appropriate circumstances, as well as coordinate that recovery.

5.2.3 An Implementation Architecture Enabling Fault-Tolerance Activities

The third and final solution principle is that the implementation architecture support the activities of fault tolerance addressed in this work: high-level error detection and coordinated error recovery. This solution principle derives from the solution strategy and addresses the requirements in the implementation architecture. There are also two different levels of the implementation architecture that must be addressed, the node level and the system level. This subsection explores both activities at both levels of the implementation architecture.

Node-Level Architecture

The most obvious architectural requirement that must met at each node is that the node architecture support the provision of the various alternate or degraded service modes associated with each fault. The software that implements alternate service is provided by application or domain experts, and the details (functionality, performance, design, etc.) of this software are not part of the approach being outlined here. In practice, the organization of the software that provides alternate or degraded service is not an issue either. The various alternate modes could be implemented as cases within a single process or as separate processes, as the designer chooses.

In order to support reconfiguration to an alternate service for error recovery at each node though, the node architecture must provide a certain set of capabilities, or services. These critical services provide the basic support needed for reconfiguration, and they are available with every process. Given a process that provides these critical services, the fault-tolerance specification need not be concerned with the idiosyncrasies of individual process functionality. As an example of critical service, consider the obvious implementation requirement that some processes in a system undergoing reconfiguration for error recovery will need to be started and others stopped. Thus, the critical services that these

processes must provide are the ability to be started and the ability to be stopped, both in a manner that safely preserves the process state in the context of the application. Neither of these actions is trivial, in fact, and neither can be left entirely to the basic services of the operating system.

Another critical service that each node must provide to support reconfiguration is the ability to switch to an alternate mode of functionality, as specified by some parameter. Often it will not be required that a process terminate completely and a new one start: the same process can be designed to provide different modes of functionality and to support switches between modes.

Processes that provide these critical services enabling recovery are called *reconfigurable processes*. An issue of concern with these reconfigurable processes is determining the list of critical services that must be supported to enable reconfiguration. However, the focus of the research is not concerned with any specific capability, such as checkpointing. For example, reconfigurable processes should conceptually be capable of and prepared to establish and discard recovery points, but this work will not demonstrate that particular capability. Rather, this research will focus on other more general capabilities required for reconfiguration. The critical services might be conceptually simple in many cases but this simplicity is deceptive. Many application processes will include extensive functionality, but this functionality does not necessarily accommodate services such as process suspension. Integration of critical service functionality with standard application functionality in the reconfigurable process is also an issue.

In addition to the reconfigurable processes in the node architecture, there must be support for two interfaces: (1) to provide error detection information from the application, and (2) to control the application and effect recovery. These interfaces to the application and reconfigurable processes are explored as part of the node implementation architecture supporting fault tolerance.

System-Level Architecture

The critical services provided by a reconfigurable process are implemented by the process itself in the sense that the service is accessed by a remote procedure call (or similar), and a mechanism internal to the process implements the service. The exact way in which the implementation is provided will be system specific.

There will be certain error recovery activities, however, that require actions external to the reconfigurable process(es). For example, when reconfiguring in response to some system error, it might be required that two processes switch to their alternate services at the same time. Another case occurs in coordinated checkpointing where the establishment of a recovery line requires multiple processes to checkpoint in synchrony.

These services point to a set of capabilities required to coordinate and control the error recovery activities of multiple reconfigurable processes, and thus these cannot be provided as critical services within any single process. To provide these capabilities, a system-level architectural component called the *Coordinated Recovery Layer* is introduced.

Again, determination of the error recovery services to be provided by the Coordinated Recovery Layer is an issue in this work, as well as mechanisms for implementing those

services. The Coordinated Recovery Layer focuses on general coordination capabilities, such as inter-process synchronization and communication. While it is envisioned that the Coordinated Recovery Layer will be an ideal place to implement functionality supporting coordinated checkpointing and recovery lines, specific activities such as that are a major area of study in their right and not the focus of this work.

In addition to the system-level construct for coordinating error recovery, a supplement to the application in the system architecture is required to perform high-level error detection. This architectural supplement for error detection is called the *Control System*, because this system effects control on the application through sensing of state and actuating of responses (much like a classical process control system). The introduction of a control system into critical applications has been studied previously [75]. This dissertation focuses specifically on the issues involved in the use of a control system construct to perform high-level error detection and effect recovery.

5.3 Solution Overview

Many results have been achieved in survivability and related fields (as will be presented later in the chapter on related work). However, existing technology does not address the key characteristics of critical information systems and the focus on relevant fault types that were discussed previously. This section describes a solution direction for fault tolerance in response to non-local, catastrophic faults in critical information systems. The solution direction is based on and extends previous work in the areas of fault tolerance, formal specification techniques, and reconfigurable distributed systems.

As mentioned previously, this research does not address detection of and recovery from *local* faults that affect a single hardware or software component. Error detection for local faults is a rich research area in its own right, and it is assumed that existing mechanisms will be in place to perform low-level error detection (e.g., intrusion detection systems for local security attacks). Similarly, it is assumed that recovery from all local faults are dealt with by some mechanism that masks their effects, as discussed in Section 4.2.

This research addresses *non-local*, catastrophic faults through application reconfiguration. Figure 4 presents an overview of the solution approach, called the RAPTOR (Reconfiguring Application Programs To Optimize Recovery) System for fault tolerance. In order to achieve this type of fault tolerance in critical information systems, the RAPTOR methodology is a specification-based approach: a formal specification defines the fault-tolerance requirements, including descriptions of the fault-tolerant system, the faults with which the system is concerned, and the application responses and reconfigurations in response to those faults. The use of a formal specification enables a synthesizer, the Fault Tolerance Translator, to generate portions of the implementation effecting high-level error detection and error recovery. In addition to the specifications, input to the translator includes a special type of process—a reconfigurable process—that supports the application reconfigurations described in the specification. The implementation architecture consists of these reconfigurable processes, the synthesized code produced by the translator, an augment to the application—the Control System—that performs high-level error detection

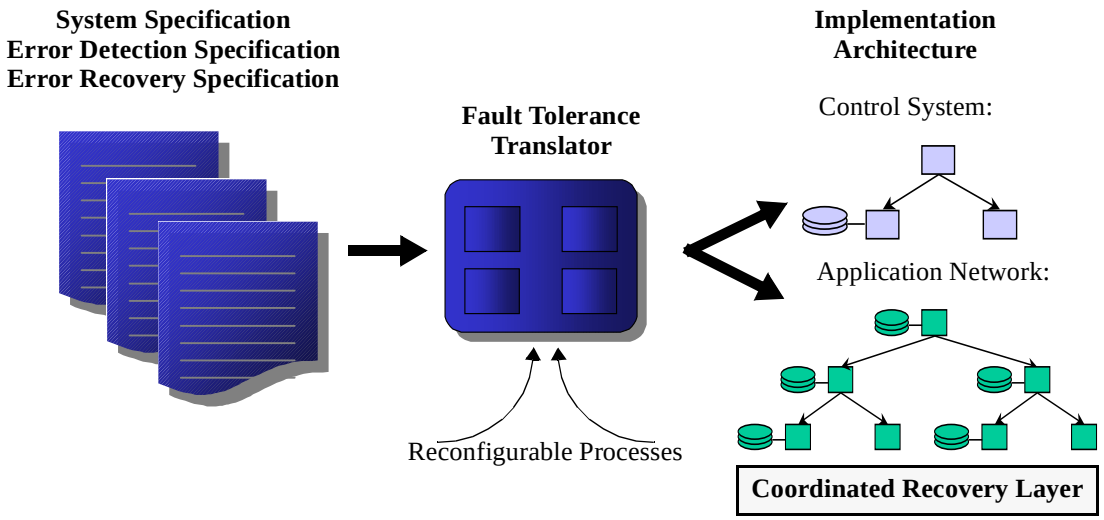

Figure 4: Solution overview—RAPTOR System

and directs error recovery, and a support infrastructure—the Coordinated Recovery Layer—that provides services to the reconfigurable processes.

The major solution components and issues relating to each aspect of the solution are described in subsequent dissertation chapters:

- *Specification of Fault Tolerance.* Chapter 7 explores the issues involved in specifying fault-tolerance requirements. The RAPTOR specification consists of three components to describe the different aspects of the problem: the system, high-level error detection, and error recovery activities. The notations utilized for each specification component and the structure of the various specifications are presented in this chapter.
- *Synthesis of Fault-Tolerance Implementation Components.* Chapter 8 explores the issues involved in synthesis of implementation components to effect high-level error detection and error recovery. The RAPTOR System provides a synthesizer to process the formal specification notations, the Fault Tolerance Translator. The specific implementation components that are generated and the process by which the translator achieves synthesis are presented in this chapter.
- *Implementation: Node and System Architecture.* Chapter 9 explores the issues involved in an implementation architecture to support error detection and error recovery. At the node level, each node must be constructed such that it supports reconfiguration and incorporates the generated code to effect reconfiguration. At the system level, a control system to perform high-level error detection augments the application architecture, and coordination and control services are provided to the reconfiguring nodes by a global entity called the Coordinated Recovery Layer.

In addition to these chapters presenting the various aspects of the solution in more detail, the next chapter (6) explores two systems built for problem investigation. These preliminary solution approaches helped clarify the requirements of a solution and refined the RAPTOR solution approach presented in the subsequent chapters.

6

Problem Analysis

Given the solution requirements, principles, and overview presented in the previous chapter, further analysis and investigation of the problem was required to refine the solution approach. This chapter discusses those efforts and the lessons learned, and the resulting solution attempts are presented in the appendices.

Throughout this chapter it is important to keep in mind that the solution approaches presented are *preliminary* attempts, and *not* a part of the RAPTOR System that is the overall solution approach presented in this work. These preliminary attempts, referred to for simplicity as STEP and PARS, investigate different aspects of the overall problem and helped to refine the ideas for the final solution approach, but both are distinct from the RAPTOR approach to fault tolerance (described subsequently in chapters seven through nine). These preliminary approaches are included in order to demonstrate and investigate issues in the problem of specifying and achieving fault tolerance.

6.1 Example System: 3-Node Banking Application and STEP

Because the solution strategy focuses on formal specification as the basis for the approach to fault tolerance, the first set of investigative activities explored what needs to be specified and what notations could be used for those specification components. To illustrate some of the issues that arise in specifying fault tolerance, consider an extremely simple example that is part of a hypothetical financial network application.

6.1.1 Example Application

The system architecture, shown in Figure 5, consists of a 3-node network with one money-center bank (*N1*), two branch banks (*N2* and *N3*), and three databases (*DB*), one attached to each node. There are two full-bandwidth communications links (*L1* and *L2*) and two low-bandwidth backup links (*l1* and *l2*). There is also a low-bandwidth backup link

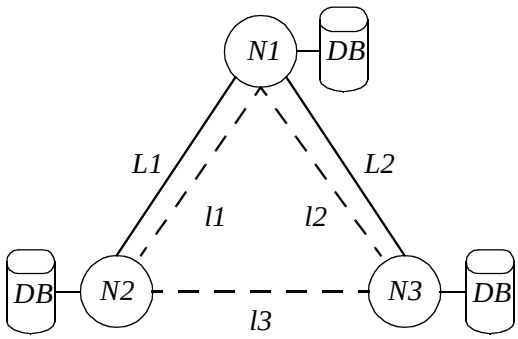

Figure 5: Example fault-tolerant distributed system

between the two branch banks (*l3*). The intended functionality of this system is to implement a small-scale financial payments system, effecting value transfer between customer accounts. In a system free of faults, the branch banks provide customer access (check deposit facilities) and local information storage (customer accounts), while the money-center bank provides branch bank asset management and routing capabilities for check clearance.

The faults with which one would be concerned in a system of this type would be the loss of a computing node's hardware, the loss of an application program on a node, the loss of a database, the loss of a communications link, and so on. For each of these, it would be necessary first to define the fault and then, for each fault, document what the system is to do if the fault arises. In this system, for example, losing the money-center bank would severely limit customer service since a branch bank would have to take over major services using link *l3* for communication. Loss of either of the full-bandwidth communications links would also drastically cut service since communication would have to use a low-bandwidth link.

To implement a fault-tolerant system, the application must be constructed with facilities to tolerate the effects of the particular faults of concern. In the system architecture, the three low-bandwidth communications links provide alternate service in case of failures in the full-bandwidth communication links (*L1* and *L2*) or the primary routing node (*N1*). The applications themselves must also provide alternate services in case of certain faults. For example, while the primary functionality of the money-center bank *N1* is to route deposited checks for clearance and maintain the balances of each branch bank, additional services that can be provided include buffering of check requests for a failed branch bank or acceptance of checks for deposit if both branch banks can no longer provide this service. Similarly, the branch banks can be constructed to provide alternate service modes, such as the buffering of check requests in case of failure at the money-center bank, or buffering of low-priority check requests in case of failure of the full-bandwidth communications link.

As mentioned previously, dealing with particular faults is only a small part of the problem. In practice, it is necessary to deal with fault *sequences*, e.g., the loss of a communica-

tions link when the system has already experienced the loss of a node. In a large infrastructure network application, there are so many components that faults arising in sequence are a distinct possibility merely on stochastic grounds. However, cascading failures, sequenced terrorist attacks, or coordinated security attacks all yield fault sequences with faults that are potentially related, e.g., all links from a node are lost in some sequence or all the nodes in a geographic region are lost.

6.1.2 Preliminary Specification Approach: STEP

Precise specification of fault tolerance in a critical information system is a complex undertaking, requiring description of both the faults that must be handled and the reconfigurations that the application performs in response to those faults. A specification of the application reconfigurations first requires that the specification capture the relevant characteristics of the application itself, including two primary components:

- The topology of the system and a description of the architecture and platform
- An abstraction of the services each node supplies to the system and the mapping of these services (including degraded and alternate services) to the platform

These two elements of the system description are necessary because a specification of reconfiguration requires description of an existing configuration and the changes to that configuration in those terms. The faults and responses comprise a portion of the fault-tolerance specification describing the necessary state changes from any acceptable system configuration to any other in terms of topology, functionality, and assignment of services to nodes in cases of the various faults of concern.

The various circumstances of interest can be described using a finite-state machine where each state is associated with a particular fault sequence. A finite-state machine is an obvious abstraction for the states associated with errors, because errors (more precisely, erroneous states) correspond to system states of interest. State changes caused by faults become inputs to the finite-state machine causing transitions; thus, sets of transitions describe the fault sequences leading to each erroneous state. Finite-state machines also provide a systematic and familiar structuring mechanism for system state, fault sequences, and responses.

In addition to enumerating the states and associated state transitions associated with the faults that can arise, it is necessary to specify what has to be done on entry to each state in order to continue to provide service. Thus, application-related actions have to be defined for each state transition, and the actions have to be tailored to both the initial state and the final state of the transition. Wide-area power failure has to be handled very differently if it occurs in a benign state versus when it occurs following a traumatic loss of computing equipment perhaps associated with a terrorist attack.

A preliminary specification approach was designed in order to explore a specification of fault tolerance for this example system. The approach, called STEP (Specification of the Three-node Example Program), consists of three components, based on the preceding discussion:

- *System Architecture Specification (SAS).* This specification component describes the system topology in terms of nodes and links.

- *Service-Platform Mapping Specification (SPMS).* This specification component relates the names of programs to the node names described in the SAS. The program descriptions in the SPMS include the services that each program provides, including alternate and degraded service modes.
- *Error Recovery Specification (ERS).* This specification component characterizes the faults of interest and their requisite responses in the form of a finite-state machine.

The ERS uses the SAS and the SPMS to describe the different system configurations, including active service modes, as states in the finite-state machine. Transitions are enumerated with faults and show the state transitions for each fault from every relevant state. The actions associated with any given transition could be extensive because each action is essentially a high-level program that implements the error recovery component of the full system survivability specification. The full specification enumerates the different states (system environments) that the system can be in, including the errors that must be detected and handled. The ERS takes this list of system states and describes the actions—i.e., reconfigurations—that must be performed when the system transitions from one environment to another.

6.1.3 Example STEP Specification

For this simple 3-node example, a prototype fault-tolerance specification was constructed using the STEP approach to explore some of the issues involved in describing a fault-tolerant system. The specification can be found in Appendix A.

The first part of the specification is a description of the system itself. Two aspects of the application are described: the system architecture and the functionality (or services) provided by the system components. The System Architecture Specification consists of a listing of nodes (including attached databases) and connections (Section A.1). The functionality of each system component in the Service-Platform Mapping Specification is a listing of different services provided by each component of the system architecture, including alternate services available in case of various system failures (Section A.2). These two specifications providing system description are rudimentary but provide a basis for specification of the various system states that arise in the event of failures.

In the STEP System Architecture Specification for this simple 3-node example, there are eleven components that can fail: the three nodes, three databases, and five links between nodes. Failures in any of these components would lead to faults at the system level; those faults would affect the service(s) provided by the particular component. In total, there are twenty-four services, some alternate or degraded, described in the STEP Service-Platform Mapping Specification. The money-center bank node provides five possible services (including its database service), each branch bank node performs seven different services, and each link provides a single transmission service. In the completely-operational initial state, there are ten services among the three nodes, their databases, and two primary links necessary to effect the fully-functional financial payments system.

The second part of the specification, the STEP Error Recovery Specification, is a description of the finite-state machine (Section A.3) and associated transitions (Section A.4) for the system and the fault sequences that are of concern. The initial state of

the finite-state machine consists of a list of the services that are operational in the case of a fully functional application. Transitions from this initial state are caused by faults, which manifest themselves as failures in one or more of the operational services. A high-level description of the fault that causes each transition is contained in the state description. In the finite-state machine transition section, a list of response activities is associated with each transition to attempt reconfiguration for the continued provision of service, as well as a list of activities to be undertaken upon repair of the fault.

In the example system, from the initial state there are eight possible faults causing transitions to other states. From those eight single-fault states, there are seventy-nine possible states should another fault occur. The finite-state machine in Appendix A only describes two sequential faults for this system, but it describes all possible two-fault sequences. The complete finite-state machine enumerating all the states associated with the various possible fault sequences would have hundreds of states, even for a simple application system such as this.

6.1.4 Discussion

In order to help evaluate the STEP specifications for this example system, a model implementation was constructed using the specifications described above. The specification provided a systematic structuring mechanism for the fault-tolerance requirements of the example application when constructing implementation of this example system.

The difficulties in achieving survivability however, even in a system as simple as this example application, are clear. The first challenge lies in describing the relevant parts of the application, the system architecture and system functionality. Then, both the initial configuration and the changes to the system configuration in terms of that system description must be specified. The problem with state explosion and the impracticality of attempting to describe the finite-state machine for a large network application is immediately obvious from the complications in this trivial system.

As seen in the motivating example, even for a simple system a specification of fault tolerance can become very large and unwieldy. Three observations can be made to deal with the specification size and state explosion problems:

- The specification structure must consist of multiple sub-specifications for describing the various components of the fault-tolerance solution, e.g. the relevant system characteristics and the finite-state machine. These sub-specifications must be integrated to describe the overall fault-tolerance solution, but the use of multiple sub-specifications enables different notations to be utilized and optimized for the particular aspect of the solution being addressed.
- The specification notation must be enhanced to accommodate larger numbers of nodes. One way of achieving this would be to introduce and integrate some form of set-based notation to enable description and manipulation of large numbers of nodes simultaneously.
- The specification itself must be constructed in such a way as to keep it manageable: for example, portions of the system should be abstracted and consolidated into single objects in the specification, thus ensuring that the specification deals with and manipu-

lates small numbers of objects regardless of how many actual nodes there are in the system.

These observations concerning the 3-node example and the STEP preliminary specification approach are helpful in development of a solution approach to fault tolerance in real systems. Using the observations and experiences from the 3-node example, a second solution attempt, called PARS, was designed to specify and achieve fault tolerance.

6.2 Example System: PAR System

This section describes a solution attempt at specifying and achieving fault tolerance called the PAR (Preliminary Attempt at RAPTOR) System. PAR expands on the STEP specification notations presented in the previous section, while also providing a preliminary synthesizer and simple implementation architecture.

Again, please note that the PAR System is *not* a part of the final solution approach, the RAPTOR System, presented in subsequent chapters. The PAR System is a prototype solution attempt, presented to investigate in more detail the problem and requirements of a solution.

This section describes the PAR solution approach first, then explores the specifications and implementation of another example system, and concludes with a discussion of observations and experiences. Appendix B presents the PAR specification system and an example application specification.

6.2.1 PAR Solution Approach

The PAR System focuses primarily on the specification notations necessary for describing fault-tolerance requirements and activities. However, to help evaluate the utility of the specification, a synthesizer was constructed that generates key code components from the specifications and a simple implementation architecture was devised that incorporates the synthesized code. All these aspects of the solution approach are discussed in this subsection.

PAR Specification Notation

The first issue in specifying fault tolerance is determining the particular components required in a specification language. From the simple 3-node system described in the previous section, one can see that a finite-state machine must be constructed to specify (1) the initial configuration of the system and (2) any reconfigurations required to recover from system errors. This finite-state machine (contained previously in the Error Recovery Specification) consists of all possible states the system could be in given the system errors that should be handled, and the activities to undertake on transition from one system state to another. Two key aspects of the system must be described: the system architecture and the services that the nodes of the system provide.

In addition to three specification components presented previously (SAS, SPMS, and

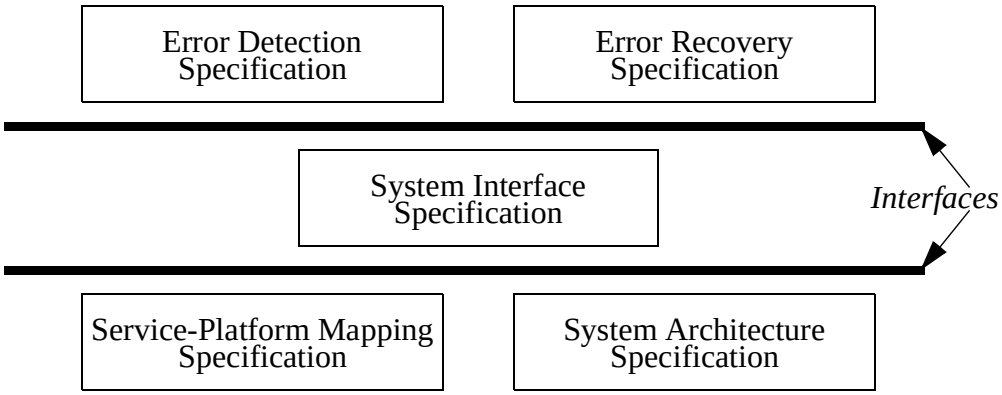

Figure 6: Multi-level PAR specification structure

ERS), another key component of the problem that must be specified in more detail is the errors of interest and the manner in which they are to be detected. In that simple 3-node system, there was no complexity to the errors: each error was the result of a component failure and affected some set of application services. It was difficult to correlate failures because there weren't very many of them. In real systems, however, one must address more than just low-level faults. The possibility of correlated low-level faults that indicate a more complex, non-local fault is very real, e.g., multiple, widespread local intrusion detection alarms over some period of time pointing towards a coordinated security attack. In order to describe these more complex, widespread, and non-local faults, another specification component for error detection is required.

The PAR specification notation involves five major sub-specifications, three of which are components (some enhanced) from the STEP approach. The structure of a PAR specification is shown in Figure 6, and the components are the following:

- *System Architecture Specification (SAS)* describes the topology of the system and platform in terms of the computing nodes and parametric information for key node characteristics. Nodes are named and described additionally with node type and any other property information required, such as geographic region, hardware details, operating system, software versions, and so on. In addition, the low-level events or errors that can occur at nodes are defined.
- *Service-Platform Mapping Specification (SPMS)* relates the names of programs to the node names described in the SAS. The program descriptions in the SPMS include the services that each program provides, including alternate and degraded service modes.
- *System Interface Specification (SIS)* defines major system objects—sets of nodes—in terms of the lower-level entities contained in the SAS and SPMS.
- *Error Detection Specification (EDS)* defines the overall systems states that are associated with the various faults of interest, in terms of the low-level events (outlined in the SAS) and combinations of those events occurring in nodes or sets of nodes defined in the SAS and SIS.
- *Error Recovery Specification (ERS)* defines the necessary state changes from any

acceptable system reconfiguration to any other in terms of topology, functionality, and assignment of services to nodes. The structure of the ERS is that of a finite-state machine, with transitions defined for occurrence of errors in the EDS.

Given the number of states that a large distributed system could enter as a result of the manifestation of a sequence of faults, it is clear that some form of accumulation of states or other simplification must be imposed if an approach even to specification of fault tolerance is to be tractable. The key to this simplification lies in the fact that many nodes in large networks, even those providing critical infrastructure service, do not need to be distinguished for purposes of fault tolerance. In the banking application domain, for example, it is clear that the loss of computing service at any single branch is both largely insignificant and largely independent of which branch is involved. Conversely, the loss of even one of the main Federal Reserve computing or communications centers would impede the financial system dramatically—some nodes are obviously much more critical than others. However, the loss of 10,000 branch banks (for example, because of a common-mode software error) would be extremely serious—even non-critical nodes have an impact if sufficient of them are lost at the same time.

To cope with the required accumulation of states, the overall specification is multilevel, and the fifth element, the System Interface Specification, is added to the specification approach. The SAS and the SPMS are declarative specifications, and in practice the contents of these specifications are databases of facts about the system architecture and configuration. The EDS and ERS are algorithmic specifications—they describe algorithms that have to be executed to perform error detection and error recovery respectively. In principle, these algorithms can be written using the information contained in the SAS and SPMS. But it is precisely this approach that contributes to the state explosion in specification. Working directly with the SAS and SPMS describing these systems leads to specifications that are just too big. For this reason, the SIS defines major system objects—sets of nodes—for manipulation in the EDS and ERS.

The overall structure of the ERS is that of a (traditional) finite-state machine that characterizes fault conditions as states (defined in the EDS using sets) and associates the requisite responses to each fault with state transitions. The fault conditions of concern for a given system are declared and described in the EDS. Arcs in the ERS finite-state machine are labeled with these fault conditions and show the state transitions for each fault from every relevant state. The actions associated with any given transition are in the ERS and are extensive because each action is essentially a high-level program that implements the error-recovery component of the full system fault-tolerance specification. The complete fault-tolerance specification documents the different states (system environments) that the system can be in, including the errors that must be detected and handled. The PAR specification notation's ERS utilizes a notational construct designed to describe the finite-state machine of the system through all relevant system errors. The notational construct for the finite-state machine enables brute-force description of all possible relevant failures in all possible states as well as the responses to those failures.

In summary, a PAR specification consists of five components corresponding to the five sub-specifications outlined above. An example fragment of a PAR specification is shown in Figure 7. This example is incomplete and uses comments for simplicity, but it illustrates

```
PAR-SAS:                    -- System architecture specification
   -- Every node, node type, and event declared; examples:
   TYPE federal_reserve; TYPE money_center; TYPE branch
   EVENT security_attack;

PAR-SPMS:                   -- Service platform mapping specification
   FORALL branch                -> customer_service,
                                   local_payment;
   FORALL money_center          -> customer_account_management,
                                   regional_payment;
   FORALL federal_reserve       -> member_bank_account_management,
                                   national_payment;

PAR-SIS:                    -- System interface specification
   -- Set declarations and definitions
   SET FederalReserveBanks; SET MoneyCenterBanks; SET BranchBanks;
   FederalReserveBanks  =  { frb1, frb2, frb3 }
   MoneyCenterBanks     =  { i : NODE | money_center(i) }
   BranchBanks          =  { i : NODE | branch(i) }

PAR-EDS:                    -- Error detection specification
   -- declare and define this attack to be more than 50 branches or
   -- all money centers or 1 Federal reserve bank detects an intrusion
   ERROR CoordinatedAttack;
   CoordinatedAttack    =
         {card(security_attack(BranchBanks)) > 50 } OR
         { FORALL i IN MoneyCenterBanks | security_attack(i) } OR
         { EXISTS i IN FederalReserveBanks | security_attack(i) }

PAR-ERS:                    -- Error recovery specification
   CoordinatedAttack:
      BranchBanks           -> customer_service.terminate;
                               local_payment.terminate;
                               local_enquiry.start;
      MoneyCenterBanks      -> customer_account_management.terminate;
                               regional_payment.terminate;
                               commercial_account_management.start;
      FederalReserveBanks   -> member_bank_account_management.terminate;
                               national_payment.limit;
```

Figure 7: Skeleton PAR specification

some of the material needed to define a wide-area coordinated security attack on the banking system and a hypothetical response that might be required.

PAR Translator

A translator was constructed for the PAR specification language that processes all five sub-specification notations and generates C++ code for actuators of the nodes in the implementation. This PAR Translator is constructed from a grammar with 73 productions and 31 tokens (presented in Appendix B). It contains facilities for simple set enumeration

and composition, Boolean logic, and quantifiers. The translator generates code on a per-node-type basis for all types declared in the System Architecture Specification.

The grammar for the PAR language parses a version of the PAR sub-specification languages in a particular order: SAS, SPMS, SIS, EDS, and ERS. In the SAS, all of the nodes are declared first, followed by declarations of node types, properties, and events. Then, a list of propositions states facts about the nodes, such as assigning types and properties to nodes. In the SPMS, service names are assigned to the different node types. In the SIS, the sets to be used are declared and then the sets are enumerated, using either explicit set declaration or set composition across node type or property. In the EDS, the system errors are declared and then described in terms of events in individual nodes or quantified across sets. Finally, in the ERS, the finite-state machine is described using the errors declared in the EDS, and the actions to be taken on state transitions are specified.

Implementation Architecture

The code generated by the PAR Translator is for actuators, the code components that receive commands to reconfigure application nodes. The implementation architecture must be constructed to accommodate and make use of the generated actuator code. Firstly, the application nodes must support the various modes of functionality described in the Service-Platform Mapping Specification. Then, integration of the actuators into application nodes involves implementing the interface for receiving a command message to start or stop a particular service and making the appropriate application call.

In order for the actuators to be utilized for error recovery, there must be a system-level construct that sends the command messages to the actuator. As discussed in Chapter 5, the augment to the application that performs high-level error detection and prompts error recovery is the Control System. In the PAR implementation architecture, the Control System must be constructed by hand, but the specifications describe the actions that must be performed to effect control.

6.2.2 Example Application

To explore the methodology presented in PAR System, a 103-node financial payments system was constructed. A specification of the 103-node banking system was designed using the five specifications notations of the PAR System, then the PAR Translator generate actuator code that was integrated into an implementation model of the 103-node system.

The PAR specification of this system can be found in Appendix B (Section B.2). The specification begins with the system description: each of the 103 bank nodes is declared with a unique identifier in the System Architecture Specification. Then, three different bank types, three events, and eight properties are declared, corresponding to geographic regions. After the declarations, each bank node is assigned a bank type and geographic region. In this system, the three bank types correspond to the different levels of the banking hierarchy: at the top level are Federal Reserve Bank nodes, in the middle are money-center bank nodes, and at the bottom are branch bank nodes. In this example the Federal

Reserve Bank has a primary node and two backups, then there are ten money-center banks with nine branch banks each.

The Service-Platform Mapping Specification follows with an enumeration of the services provided by each node type. Nodes of type *federal_reserve* contain seven services, *money_center* nodes have twelve services, and *branch* type nodes provide eleven services. Branch bank nodes provide check deposit facilities, a database service storing customer accounts, and other services to route and service check requests; there are a total of eight services in the normal operating mode and three alternate services. Money-center bank nodes route check requests and responses, batch requests and responses to pass to the Federal Reserve, and maintain branch bank balances in their databases; eight of the twelve services are utilized in the normal operating mode, while four are provided as alternate services. Federal Reserve banks route batch requests and responses, as well as maintain money-center bank balances using a database; only three services are necessary during normal operation, and four alternate services are specified.

Next, the System Interface Specification declares the set objects of interest, and then enumerates each of those sets. Some of the sets defined correspond to the different node types: sets enumerate Federal Reserve banks, money-center banks, and branch banks. Sets also are defined corresponding the different properties, such as geographic regions. Other sets correspond to special characteristics such as whether a Federal Reserve bank is serving as a primary or backup. The sets can be enumerated either explicitly or using a set composition format; the example specification utilizes both enumeration methods.

The Error Detection Specification declares and describes the different errors of concern in the system. The errors are described using Boolean expressions consisting of set operations on the previously declared events. For this example, four errors of concern are defined: *PrimaryFrbFailure*, *McbSecurityAttack*, *CoordinatedAttack*, and *WidespreadPowerFailure*.

Finally, the Error Recovery Specification explicitly enumerates the finite-state machine for the system using the errors defined in the EDS, then describes the actions to take on each transition in the state machine. Some of the errors are parameterized, with either a set or node name, and this parameter is passed to the action definition. The action definitions use the services defined in the SPMS to direct nodes and sets of nodes to reconfigure for recovery from the system errors.

To explore the issues in synthesis, the PAR Translator processed the sample specification of the 103-node financial payments application and generated actuator code. In order to utilize this actuator code, application nodes for each of the bank types were implemented, as well as a control system to monitor and effect control. Then, the actuator code was integrated into application nodes, and a complete system was constructed, based on the PAR specifications, that demonstrated various recovery activities.

6.2.3 Discussion

The specification and implementation of the 103-node example banking system yielded many observations related to the PAR System:

- The System Architecture Specification for defining all the nodes in the system and

their pertinent characteristics is cumbersome for describing a system of any non-trivial size in any detail. Because this information most certainly exists elsewhere already (e.g., in databases), constructing an input filter to process that information directly would be a simpler way to define the architecture and characteristics of the system.

- The Service-Platform Mapping Specification uses a simple naming mechanism for defining services; this could and probably should be augmented with more formal definition of the functionality provided by each service.
- The Error Detection Specification should be utilized for synthesis of Control System implementation components. The translator processes and stores the information related to error detection; this information could be utilized in the Control System with the appropriate target for code generation.

One particular specification problem that must be explored further in the motivating example is the state explosion problem in the finite-state machine. The state explosion problem was eased by the introduction of sets into the specification: this enables the finite-state machine to refer to erroneous states and transitions involving multiple nodes, rather than complicating the finite-state machine unnecessarily with transitions for one node at a time. However, the PAR specification notation does not address this problem directly in the description of the finite-state machine in the ERS: the description still requires explicit enumeration of all relevant transition sequences. There are a number of approaches that can be employed to control the state explosion problem:

- The number of possible system states must be restricted—in systems this large and complex, some forms of abstraction must be employed to keep the number of possible system states tractable.
- The number of sequential failures that can be tolerated must be restricted—handling large numbers of sequential failures in the general case cannot be achieved in a specification of considerable size.
- If a system can be constructed such that sequential failures are handled independently of the order or occurrence of previous failures, this simplifies the specification greatly. In this case, the different failures that can occur in the system can be specified and handled without additional structuring.

These three approaches all address the complexity of the system being described, rather than providing a specification solution to help manage that complexity. On the one hand, critical information systems are *extremely* complex systems (for a variety reasons, as discussed in Chapter 2), and any ways of reducing the inherent complexity that must be handled should be employed if possible. The RAPTOR System, however, introduces mechanisms into the specification to help manage complexity and the state explosion problem. The next section discusses concepts for development of the RAPTOR System.

6.3 Concepts for the RAPTOR System

The experiences from the solution attempts presented in this chapter, STEP and PARS, point to a more refined approach for the RAPTOR System. In particular, the need to accommodate more complex, large-scale systems in specification points to the introduc-

tion of various forms of abstraction. In addition, an implementation architecture building on the abstraction of the specification is discussed.

6.3.1 Specification and Abstraction

There are various forms of abstraction that can be introduced into the specification in order to help manage complexity and state. The consideration of issues in the finite-state machine description and state explosion problem when specifying these critical, large-scale systems points towards the exploration of abstraction and hierarchy.

In the PAR System, the System Interface Specification enabled description of sets of nodes, thus introducing an abstraction mechanism to allow definition of an object referring to multiple nodes simultaneously. Being able to refer to many nodes simultaneously, such grouping nodes with certain characteristics such as common administrative domains or geographic regions, can reduce the size and complexity of a specification. The hierarchic nature of many information systems leads to the notion of groupings of local nodes according to various pieces of data. Sets of local nodes can be grouped to form regional sets based on common data. Data on regional sets of nodes can be referenced at higher levels of abstraction, larger groupings of nodes up to a system or global level. The hierarchical abstraction of data in a system enables management of complex system descriptions.

This hierarchical abstraction of system data can be applied to faults—i.e., events—and error detection within the system as well. Events occurring at the local level can be organized into a finite-state machine at local nodes to handle faults and manage state at the lowest level of the system. Events can also be passed up (and down) a hierarchy of finite-state machines, so that a collection of local events can be recognized at the regional level as a regional event, and regional events could be passed up further to recognize system-level events.

Finally, hierarchical abstraction also applies to responses to faults—actions—in the system. Responses to local faults often occur only at the local level, but some local faults require regional activity, and vice versa. Similarly, regional recovery actions can sometimes cause some global response to occur.

Figure 8 shows this notion of hierarchical abstraction, represented as occurring in a vertical direction. While the PAR System accommodated some form of abstraction in terms of data (grouping similar nodes), the provision for hierarchical abstraction throughout the specification of fault tolerance helps manage complexity and state in the RAPTOR specification method (discussed in the next chapter).

There is a second, complimentary form of abstraction required for specification of fault tolerance, as well. Represented in the horizontal direction in Figure 8, there is a flow of information occurring from data to events to actions at each level of the hierarchy. For example, at the local level, data about the local node determines the events (faults) that can occur at that level; then, the actions in response to those events are defined for the local node. Similar abstractions occur at regional and system (or global) levels.

This second abstraction mechanism can be used to help control the state explosion problem. It should be possible to construct finite-state machines at the node or local level

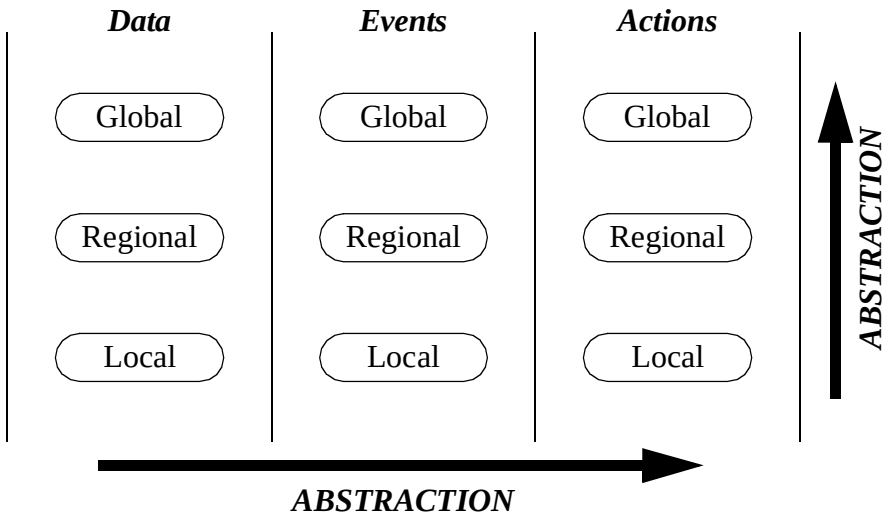

Figure 8: Two directions of abstraction

to control error detection and error recovery internally, and only pass data, events, or actions up to the regional level in circumstances where the situation calls for that. Likewise, the system or global level state machine can be simplified by handling regional data, events, and actions at the lower level whenever possible. This separation of concerns—defining data, events, and actions in each level of the hierarchy—enables faults to be described and handled at the appropriate level of interest, thus simplifying the specifications of fault tolerance in complex systems.

The introduction of these forms of abstraction into the RAPTOR System helps manage complexity and state in the specifications and is discussed further in the next chapter.

6.3.2 Implementation Architecture

The abstraction and hierarchy introduced for the RAPTOR specification structure points towards a refined implementation architecture supporting fault-tolerance activities. The introduction of hierarchic finite-state machine descriptions for data, events, and actions requires finite-state machines at the levels of concern (local, regional, global) to effect control at each level. This impacts the architecture of the Control System, requiring error detection and error recovery at various abstraction levels. Sensors at application nodes must still communicate event information to the Control System, but that event information must reach the appropriate level of the finite-state machine hierarchy. Similarly, finite-state machines in the Control System must communicate recovery actions to the appropriate application node actuators. These issues are discussed in more detail in Chapter 9.

6.3.3 Summary

The experiences in specification of the 3-node example and construction of the PAR System contributed to a better understanding of issues involved in specification and implementation of fault tolerance. The next three chapters present the details of the RAPTOR System, the solution approach to fault tolerance in critical information systems.

7

RAPTOR Specification

The preliminary investigation into solution directions presented in the previous chapter refined the ideas and led to development of the RAPTOR specification system. This chapter describes in detail the components, notations, and structure of a RAPTOR specification for fault tolerance.

7.1 RAPTOR Specification Components and Notations

The preliminary specification attempt in the PAR System (Section 6.2.1) contained five sub-specification components: the System Architecture Specification (SAS), Service-Platform Mapping Specification (SPMS), System Interface Specification (SIS), Error Detection Specification (EDS), and Error Recovery Specification (ERS). The five components were integrated using a single, custom-designed notation. The overall approach had its shortcomings, as discussed previously (Section 6.2.3). The RAPTOR System presents a more refined specification approach based on a better understanding of the problem and requirements.

Firstly, while the five PAR specification components describe all the aspects of the problem, in actuality there are only three distinct elements of the fault-tolerance specification: (1) the system, (2) the errors to be detected, and (3) the responses to those errors. In the first version of the specification approach, three different components comprise the system description: the SAS, SPMS, and SIS. (The errors to be detected and the error responses correspond one-to-one with the EDS and ERS, respectively.) While all three specification components were required to describe different aspects of the fault-tolerant system in that solution approach, an ideal solution would be a notational approach that integrates the entire system description. With that in mind, the RAPTOR System for specification has only three components:

- System Specification
- Error Detection Specification
- Error Recovery Specification

Secondly, each specification component in the RAPTOR System should utilize a notation best suited to that which is being specified. It is unreasonable to think that the same notation used to describe application functionality—custom-designed or otherwise—would be optimal for specifying a finite-state machine description of high-level errors and responses. The following subsections present the notations that the RAPTOR System utilizes for each of the three specification components.

7.1.1 System Specification

The first aspect of fault tolerance that must be specified is the system itself: its configuration, capabilities, state, and other pertinent characteristics. As stated previously, in order to specify the faults of concern, one must be able to refer to the component(s) of the system in which the fault exists and those components affected by the fault. Similarly, in order to specify recovery and reconfiguration activities in response to the faults of concern, the configuration of the system must be known first and the parts of the system to be reconfigured must be outlined.

The items that should be included in a system description for the purposes of fault tolerance include the following:

- Node types
- Pertinent node characteristics, such as hardware configurations, operating system versions, software configurations, and other application-specific information
- Services provided by each node, including both normal and alternate functionality
- Relationships amongst nodes, including groupings of related nodes

For large, complex systems such as critical information systems, the system description will result in an enormous specification. It is probably the case, though, that much of this information is already known and being stored in various applications and databases for a variety of purposes. For example, system administrators track hardware and software configurations throughout their networks. Similarly, monitoring software is often run on critical systems such as these to observe system state and ensure proper operation.

The key capability in this context is to maintain all relevant aspects of the system description in an easily accessible location, using a flexible notation that facilitates specification of complex systems. The RAPTOR System utilizes an object-oriented database to describe and store the pertinent characteristics of the system. An object-oriented database provides both the notational and storage capabilities necessary for system specification. A database system of some sort is required simply to cope with the sheer volume of information required to describe a critical information system; it is likely that some of the descriptions of relevant system characteristics already exist in databases. The primary advantage of an object-oriented database is that the language used for description provides encapsulation, inheritance, and other abstraction mechanisms necessary for coping with large and complex descriptions associated with critical information systems. The use of an object-oriented database language works well with object-oriented modeling techniques, which are often employed during requirements and system analysis [12]. An object-oriented notation, such as that provided by object-oriented database definition languages, contains the necessary abstraction facilities to describe complex systems like critical information

systems.

Specifically, RAPTOR uses the POET object-oriented database [59] to provide the required storage and notational facilities for the System Specification. Given that comparison and evaluation of object-oriented databases was not a major objective in this work, this particular object-oriented database was chosen for a number of pragmatic reasons:

- POET allows the specifier to use C++ as the database definition language, then POET provides a preprocessor and compiler to generate its own database schemas and database from the C++ class definition.
- The use of C++ as the database definition language is a major advantage, as the language is already familiar to many engineers and programmers and does not require the specifier to learn the idiosyncrasies of another object-oriented notation.
- POET enables easy integration of database functionality into object-oriented applications, using a variety of development environments [59].

The C++ class hierarchy used for system specification enables the multiple levels of abstraction to be easily described using encapsulation and inheritance. Additionally, arbitrary relationships, not just hierarchical structures, can be specified using this description method. Finally, multiple abstraction hierarchies can be specified, allowing nodes to belong to different groups based on general characteristics. For example, in the hypothetical banking system from the previous chapter (Section 6.2.2), one abstraction hierarchy would correspond to banking relationships and administrative control, where branch banks are owned by their corresponding commercial money-center banks, and then all the money-center banks are owned by the Federal Reserve Bank (this maps to the system architecture). Another abstraction hierarchy could be created though to represent the distribution of electric power, where power companies contain references to all the banks that they serve. Bank nodes would thus belong to both abstraction hierarchies.

7.1.2 Error Detection Specification

The second aspect of fault tolerance that must be specified are the errors to be detected. Recall that the errors of concern in this work are not just local faults or simple events; complex faults requiring correlation and combinations of events are a key concern. The error detection specification must be able to describe complex combinations and sequences of events to detect high-level errors caused by non-local faults as well as local faults.

As shown in the previous chapter, the errors to be detected can be outlined and organized into a finite-state machine. The finite-state machine is a natural description mechanism for detection of errors, because an error (more precisely, an erroneous state) corresponds to a system state of interest. As the state changes, the events that cause changes become input to the finite-state machine in the form of events; error states in the finite-state machine define the faults of interest that must then be handled and tolerated.

In complex, large-scale systems, however, a single finite-state machine to describe the overall system state would be impossibly difficult to manipulate, and at the same time would not take advantage of the natural abstraction levels within the system. The RAPTOR Error Detection Specification revolves around a collection of communicating finite-

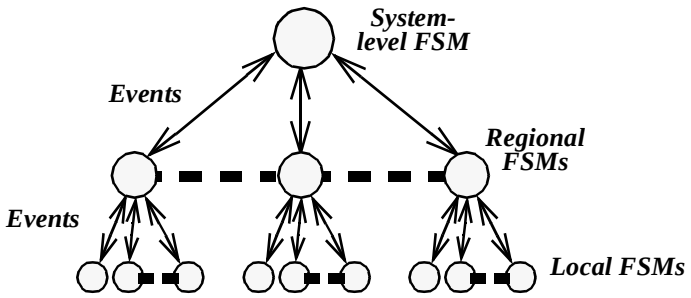

Figure 9: Network of communicating finite-state machines

state machines for each node type or collections of nodes. The communicating finite-state machines are organized into different levels or possibly hierarchically (see Figure 9), though any arbitrary structure can be described. States are defined for each node type or collection of nodes. Inputs to the finite-state machine that cause transitions from one state to another are events that occur at the node level; these events are faults within nodes and can be handled appropriately there much of the time. Additional inputs, however, are events that are communicated between finite-state machines to detect more complex faults. The network of finite-state machines works collectively to detect more significant non-local faults (possibly sequential or catastrophic) through the composition of low-level, local fault events.

For example, to detect a widespread, coordinated security attack, it is necessary to collect events from multiple lower-level nodes, recognizing the simultaneous intrusion detection alarms triggered at those nodes. Similarly, a cascading software failure can be detected by recognizing multiple software failure events occurring in the system in some predefined sequence.

The specification of finite-state machines for the Error Detection Specification suggests a state-based specification language. The mapping from finite-state machines to a state-based specification language would facilitate description of the errors to be detected as well as the overall node and system states. Ideally, the notation adopted for the specification of error detection would be formal so as to permit various forms of analysis as well as facilitate synthesis of implementation components.

RAPTOR uses the formal specification notation Z to describe the finite-state machines for error detection [27]. Z is a mathematically rigorous language based on first-order predicate logic and set theory that has been used for many years in the specification of complex systems [13], [20]. For a detailed presentation of the Z formal specification notation, please refer to texts by Diller [27] or Potter [60].

As a general-purpose, state-based specification language, Z enables communicating finite-state machines to be defined formally and in a very straightforward manner. Z uses a notational construct called a *schema* for the grouping of all relevant information in a state description. Schemas are used for specifying both states and state transitions [27]. In the RAPTOR Error Detection Specification, a Z state description schema is defined for each

finite-state machine, including any invariants and necessary state variables from the System Specification. Events for finite-state machines at each level of the abstraction hierarchy (node, region, or system) are declared as messages, and schemas for each event are defined to specify that event's effect on the state. Detailed examples of each of these concepts will be presented later in this chapter.

7.1.3 Error Recovery Specification

The third aspect of fault tolerance that must be specified is the set of recovery actions performed in response to errors. The Error Recovery Specification outlines the activities to undertake for each fault in the Error Detection Specification. Because faults correspond to the transitions in the finite-state machines, error recovery actions are defined for each transition.

As mentioned previously, the application must be constructed so as to provide for the error recovery actions required in response to the prescribed faults. The Error Recovery Specification can refer to these actions in an abstract manner, for example, by defining a set of messages that must be sent to application nodes corresponding to the recovery actions. Thus, the specification defines a high-level "program" for each set of recovery activities in response to faults.

The Error Recovery Specification also uses Z to describe the set of actions performed in response to each fault. These actions correspond to message definitions, and messages are possible outputs of the finite-state machines of the Error Detection Specification. The messages specified in the ERS correspond to actual messages that will be sent to application nodes in an operational fault-tolerant system.

7.2 RAPTOR Specification Structure

There are two parts to the structure of a RAPTOR specification: (1) the System Specification utilizing the POET object-oriented database language and (2) the Error Detection Specification and Error Recovery Specifications written in the Z formal notation.

Throughout this section an example specification will be used to present details of the specification notations and structure. The example fragments are from the financial payments system model, provided in its entirety in Appendix C. The system structure and node types are similar to the 103-node example system used in Section 6.2.2 (though the scale of the system is much larger). One type of bank, a money-center bank, is presented in the running example throughout this chapter and subsequent chapters. A money-center bank is a mid-level bank in the financial payments system hierarchy: money-center banks are connected to the Federal Reserve Bank at the top of the hierarchy, while being responsible for multiple branch banks lower in the hierarchy. Functionally, a money-center bank routes checks from its branch banks for settlement, maintains branch bank balances, batches checks destined for other money-center banks to pass to the Federal Reserve Bank, and processes batches of checks from the Federal Reserve Bank.

7.2.1 System Specification

The System Specification is a collection of object-oriented class definitions written to define the POET object-oriented database. The POET language is based on C++, supplemented by special keywords used by the POET preprocessor to construct the database schemas. Otherwise, the class definitions can make use of inheritance and encapsulation just as any C++ class definition. The class definitions consist of methods and member variables, some of which can be references to other objects in the database.

Figure 10 shows the system specification for a money-center bank. The POET keyword, *persistent*, precedes the class declaration in order for the POET preprocessor to recognize this class definition as part of the database definition. The *MoneyCenterBank* class definition inherits off of the *Bank* class definition (not pictured here, but provided in Appendix C). The Bank class definition encapsulates the methods and member variables common to all types of bank objects, including those relating to bank names and unique bank IDs. The attributes section of the class definition enables description of all the relevant characteristics of a node or object; for the *MoneyCenterBank* object this includes hardware and software platforms, but as many key elements as necessary can be included.

```
// Class definition
persistent class MoneyCenterBank : public Bank
{
   public: // Constructors & destructor
      MoneyCenterBank(const int bankId, const PtString& name);
      virtual ~MoneyCenterBank();

   public: // Services
      void Print() const;

   public: // Accessors (in base class Bank)

   private: // Attributes
      HardwarePlatform m_HardwarePlatform;
      OperatingSystem m_OperatingSystem;
      PowerCompany* m_PowerCompany;

   public: // Pointers
      FederalReserveBank* m_Frb;

   public: // Sets
      lset<BranchBank*> m_SetBbs;

   public:
      MoneyCenterBankFSM m_fsm;
};
```

Figure 10: Money-center bank system specification

The next two member variables are examples of references to other database objects; *m_Frb* points to the object in the database that refers to the parent of this money-center bank in the banking hierarchy, while *m_SetBbs* is a POET set/container mechanism holding references to the database objects for branch banks connected to this money-center bank. Finally, the last member variable refers to a finite-state machine class that can be defined for this node type and included as part of its class definition.

In summary, the System Specification consists of class definitions describing nodes and sets of nodes in the system. These descriptions include relevant characteristics and relationships between objects in the system and can be as detailed as necessary for the given application. The descriptions can later be instantiated and stored in the POET database for usage in an operational system. Two complete examples of RAPTOR System Specifications can be found in Appendix C (sections C.2.1 and C.3.1).

7.2.2 Error Detection and Error Recovery Specifications

Both the Error Detection and Error Recovery Specifications use the Z formal specification language as their notation for describing the fault-tolerance activities. More specifically, these two fault-tolerance descriptions utilize the Zeus system [77] for developing specifications with the Z notation. Zeus is a comprehensive toolset for Z that provides an integrated editing and analysis environment through the use of the Adobe FrameMaker document processor [1] and the Z/Eves theorem prover [68]. Because of the special characters required in Z, previous support for specification in Z has required using either specialized character sets that provide no integration with other Z tools, or a text-based LaTeX Z implementation [73] that is less readable than if one were able to see the actual Z characters during construction of a specification. Zeus allows the specifier to work in an intuitive, "WYSIWYG" environment, with the FrameMaker document processor providing the special character set for Z just as it would any other character set (such as mathematical symbols). Zeus then also provides a back-end to communicate with the Z/Eves tool, which provides type-checking and theorem proving for Z specifications [68]. Z/Eves happens to use the LaTeX Z language definition, so Zeus processes the Framemaker specification and outputs the LaTeX translation to Z/Eves for processing, transparent to the user. Zeus then reports back to the user any output or error messages from the Z/Eves analysis.

Given the use of Zeus (and therefore Z/Eves for type checking), the Error Detection and Recovery Specifications will have a structure imposed in order that they successfully type check. At a minimum, this means that schemas and definitions must be declared in the order of their usage. Because I provide a translator to generate code from this specification, some additional structuring rules are imposed (these will be discussed in Chapter 8).

The structure of the RAPTOR Error Detection and Recovery Specifications consists of three types of files: (1) the declarations file, (2) the state descriptions file, and (3) finite-state machine files. Figure 11 depicts the relationships between these files, as well as their usage by a synthesizer, the Fault Tolerance Translator (described in the next chapter). The

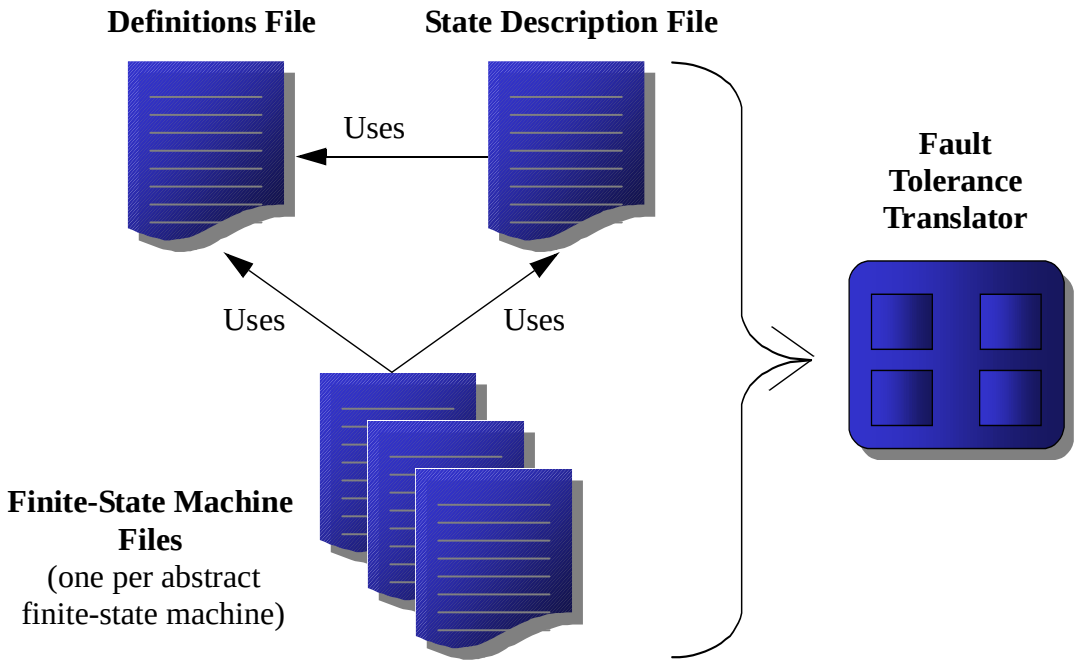

Figure 11: Files of a Z Error Detection and Recovery Specification

composition of and relationships between each of these file types will be discussed in detail in the next three subsections. Two complete examples of RAPTOR Error Detection and Recovery Specifications can be found in Appendix C (sections C.2.2 and C.3.2).

Declarations File

A RAPTOR specification for error detection and error recovery begins with a file of declarations. The first declarations will be *given sets* in Z, if any are required. Given sets are much like undefined terms in geometry; they are sets that can be utilized throughout the specification but require no further definition—it is assumed that the specifier and user understand what these sets signify.

The next section contains *axiomatic descriptions*. Axiomatic descriptions define additional variables (and their types) that are global to the specification. The most important axiomatic description is the global definition of the *Time* variable. Schemas can use the *Time* variable to specify any performance or timing-related operations. (The construction of time used in this Z specification is suggested by Spivey [72].) Any other global variables required for a particular system can be specified as axiomatic descriptions.

Next in the declarations file are set definitions. Set definitions define the set names and members that will be used throughout the specification. Two very important sets are required in the RAPTOR Error Detection and Recovery Specifications:

- *SystemEvents*. This set enumerates the events that can occur in the system, both local and those recognized and communicated between finite-state machines.

S *MoneyCenterBankFSM*

MyMcb	:	$\mathbb{N}$
MyBbs	:	$\mathbb{F}\ \mathbb{N}$
HostBbs	:	$\mathbb{F}\ \mathbb{N}$
MyFrb	:	$\mathbb{N}$
NodesDown	:	*McbNodeStates*
DbDown	:	*bool*
Alert	:	*bool*
BackupFrb	:	*McbBackupFrbStates*
BbsUp	:	$\mathbb{F}\ \mathbb{N}$
BbsDown	:	$\mathbb{F}\ \mathbb{N}$
BbsIdAlarms	:	$\mathbb{F}\ \mathbb{N}$
BbsAlarmTimes	:	$seq\ \mathbb{N}$
BbNodeStates	:	*McbBbNodeStates*
BbIdStates	:	*McbBbIdStates*
BbCoordinatedAttack	:	*bool*

HostBbs	$\subset$	*MyBbs*
BbsUp	$\subseteq$	*MyBbs*
BbsDown	$\subseteq$	*MyBbs*
$BbsUp \cup BbsDown$	$=$	*MyBbs*
BbsIdAlarms	$\subseteq$	*MyBbs*

- *MessagesToApplication.* This set enumerates the messages that can be sent to application nodes in response to system errors or other events. This list of messages defines the Error Recovery Specification. The enumerated messages correspond to the services provided by the application nodes for recovery; the application must provide recovery and reconfiguration code to respond to various events, and these message definitions signify the possible responses of which the application is capable. Each service is referred to by its corresponding message in this set, and nodes are signaled to reconfigure to a new service by sending the corresponding message from the control system to the node.

The final section in the declarations file is for two special schemas: *OutputEvent* and *OutputMessage*. The *OutputEvent* schema defines the structure of events that are communicated between finite-state machines, and the *OutputMessage* schema defines the structure of messages sent to application nodes.

State Description File

The next file in the RAPTOR specification contains the schemas for the overall state description. The state description file defines state schemas for the finite-state machine abstractions for different node and set (object) types. The state schemas consists of variables defining the components of the state for each finite-state machine object. Additionally, each state schema can contain propositions relating to the component members of the state to facilitate understanding and analysis of the specification. Lastly, the overall system state schema, which enumerates all the objects in the system, can be provided in this file (though this is not a required element).

Figure 12 shows the abstract finite-state machine definition for a money-center bank. The top half of the schema is the declarations section. Some of the declarations in this schema are references to other banks, represented as integer IDs: *MyMcb* is this node's ID number, *MyFrb* is the Federal Reserve Bank parent ID, and *MyBbs* is a finite set of IDs for the branch banks owned by this money-center bank. Other declared state variables track the status of this node: *NodesDown*, *DbDown*, and *Alert* (for response to intrusion detection alarms). Other state variables track the state of related nodes; for example, *BbsDown* and *BbsIdAlarms* hold sets of IDs for branch banks that have failed or experienced intrusion detection alarms, respectively. The bottom part of the schema, the predicate on the state, presents the state invariant.

Finite-State Machine Files

Each of the remaining files in the RAPTOR specification describes an abstract finite-state machine for a node type. The first section in these files is for declarations that are specific to this abstract finite-state machine. The next section in a finite-state machine specification file contains initialization schemas. The first initialization schema sets the default values for the state schema upon initialization. Additional initialization schemas can be defined for setting member variables to values other than their defaults. For example, any sets declared in the state schema for an abstract finite-state machine will be initially set to empty, but an additional initialization schema can be defined to add members to the set.

Following the initialization schemas in a finite-state machine specification file are the low-level event schemas, defining the changes in state upon occurrence of an event or failure. The key distinction between low-level events and high-level events is that low-level events act directly upon the state in response to an event, whereas high-level events require analysis of the state in order to detect their occurrence and then effect changes on the state (see Figure 13).

The low-level events that can be handled are defined by the SystemEvents set in the definitions file. The schema name for each low-level event must be formed by pre-pending "**Schema*" to the name of the event being handled (where * can be any valid Z string). These operation schemas define transitions for the particular finite-state machine being specified. There are three possible types of activities in these operation schemas:

- Effecting changes in the state schema for this particular finite-state machine

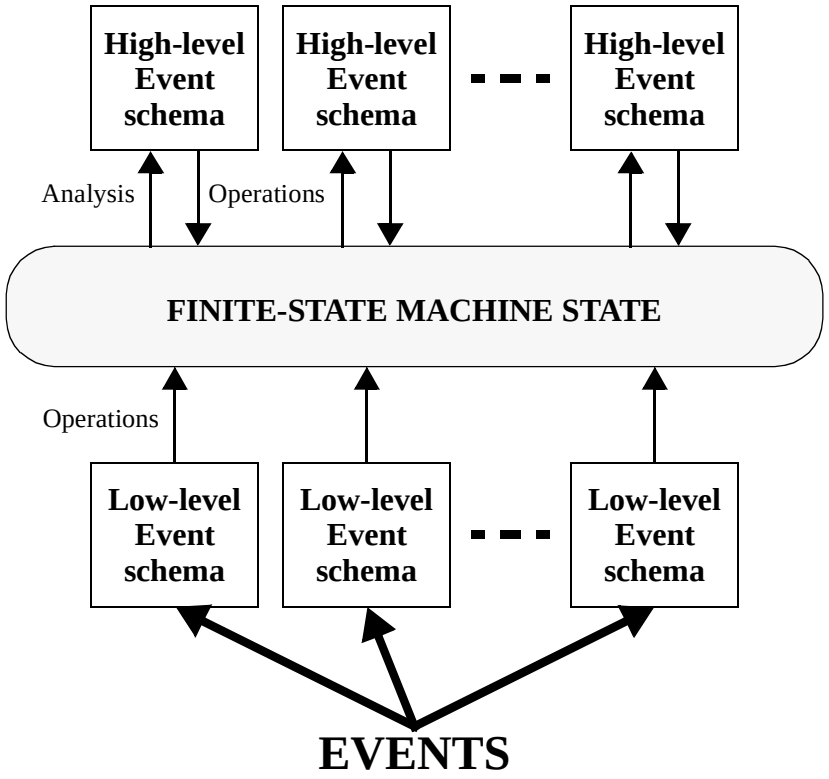

Figure 13: Low-level versus high-level events

- Generating messages to send to the application nodes in response to these events
- Generating event notifications to communicate to other finite-state machines

Figure 14 shows a low-level event schema for a money-center bank, describing what actions to take upon occurrence of a local intrusion detection alarm. In the first part of this operation schema are declarations of the state schemas that will be manipulated, including the *MoneyCenterBankFSM*. In the second part of the schema are the actions taken upon occurrence of this event: first, the state variable *Alert* is set to true indicating this event has taken place, then both an *OutputEvent* and *OutputMessage* are defined. The *OutputEvent* communicates to the parent *FederalReserveBankFSM* and the children *BranchBankFSM* schemas that this money-center bank is on alert. The *OutputMessage* indicates that the application node must be notified and initiate a new service, referred to as *McbRaiseAlert*.

The last section in a RAPTOR finite-state machine file describes high-level events in the abstract finite-state machine. High-level events are those events that are caused by changes in state variables rather than incoming events, thus requiring analysis for detection. That is the only difference between high-level events and low-level events (the fact that they are not tied to specific input events). These schemas define transitions in the finite-state machine just as before and the types of activities possible in these operation schemas are still the same. These operation schemas are denoted by the string "**Condi-*

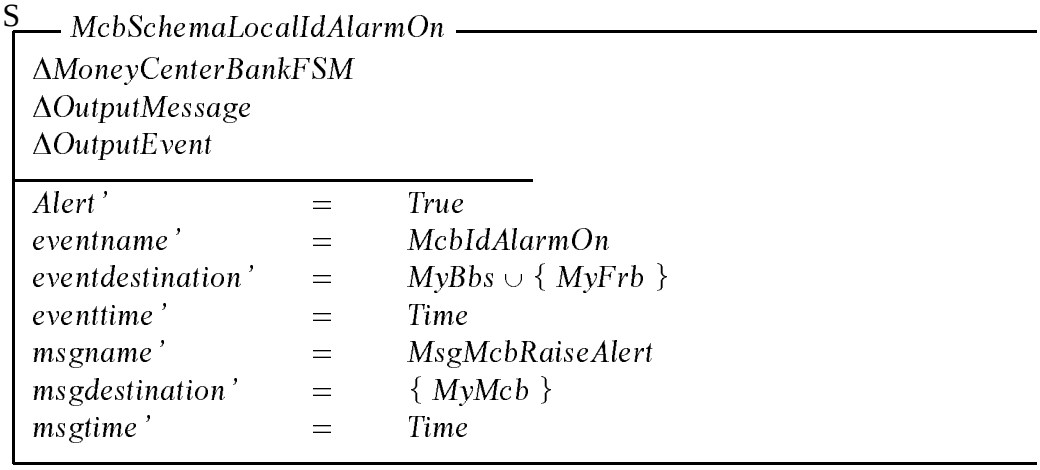

Figure 14: Money-center bank intrusion detection alarm on event

*tionSchema**".

Figure 15 shows a high-level event schema for a money-center bank, describing what actions to take upon occurrence of a coordinated security attack on the set of branch banks, defined here as when ten branch bank intrusion detection alarms have gone off in the past sixty seconds. Again, the first part of this operation schema declares the state schemas affected. The second part of this schema begins with a proposition that determines whether or not this schema comes into effect. In this case, there are three elements that must be checked:

- The state variable indicating whether a coordinated security attack is underway
- The number of elements in the sequence of branch bank alarm times (# represents cardinality in Z)
- The time for the alarm ten previous in the sequence, and whether this time is within the past sixty seconds

The proposition works as follows: if a coordinated security attack is not underway, and if at least ten alarms have occurred, then the final condition checks for the time of the alarm ten back in the sequence (an ordered set in Z). If all three conditions are true, this indicates that a low-level event, a branch bank intrusion detection alarm, has just occurred that changed the state to satisfy the conditions for a coordinated security attack. The actions to perform in this case are to modify the state variable indicating a coordinated security attack is underway and to notify the branch banks and Federal Reserve to enable proper response there.

Another important class of events that can be specified in the Error Detection and Recovery Specifications is the set of events signifying recovery from a failure. Recovery events are treated no differently from failure events, and in the example specifications in the Appendix half of the schemas are recovery operations.

$$
\begin{array}{|l}
\hline
S \quad \textit{McbConditionSchema10AlarmsIn60Seconds} \\
\Delta \textit{MoneyCenterBankFSM} \\
\Delta \textit{OutputEvent} \\
\hline
(\textit{BbCoordinatedAttack} = \textit{False}) \land (\#\textit{BbsAlarmTimes} \geq 10) \land \\
(\textit{BbsAlarmTimes}\ ((\#\textit{BbsAlarmTimes}) - 9) \geq (\textit{Time} - 60)) \\
\textit{BbCoordinatedAttack'} = \textit{True} \\
\textit{eventname'} = \textit{McbBbCoordinatedAttack} \\
\textit{eventdestination'} = \textit{MyBbs} \cup \{\ \textit{MyFrb}\ \} \\
\textit{eventtime'} = \textit{Time} \\
\hline
\end{array}
$$

Figure 15: Money-center bank coordinated security attack event

7.3 Summary

A RAPTOR specification of fault tolerance has three components: the System Specification, the Error Detection Specification, and the Error Recovery Specification. The System Specification utilizes an object-oriented database notation to describe the relevant system and application characteristics, while the Error Detection and Recovery Specifications outline fault-tolerance requirements using communicating abstract finite-state machines in the formal specification notation Z.

The System Specification enables the following types of information to be specified:

- Node types
- Node identification information
- Relationships between nodes in the system
- Node properties (such as hardware platforms, software configurations, and capabilities)
- Other relevant, arbitrary system characteristics and application properties

In addition, because the System Specification utilizes an object-oriented database notation for its description language, the specification can be stored and accessed during system operation.

The Error Detection Specification and Error Recovery Specification enable the following types of information to be specified:

- Finite-state machines representing the possible states of nodes or sets of nodes in the system
- Events that can occur within the system (including low-level and high-level errors, recovery from errors, and time-based events) that cause a change in the node and/or system state, thus signifying a transition in the abstract finite-state machine

- Communication between finite-state machines, signifying events occurring in nodes or sets of nodes that affect other finite-state machines in the system
- Messages to application nodes, representing commands to the application for changes in service or reconfiguration in response to errors
- Responses to events, both low-level errors and high-level detected errors (as well as recovery from both types of error), in terms of changes in state, communication with other finite-state machines, and messages to application nodes

It is important to note that because Z, a formal specification notation, is used for the Error Detection and Recovery Specifications, these specifications are amenable to various forms of analysis. The specifications were type-checked by the Z/Eves theorem prover, so they are syntactically correct. In addition, other types of analysis are possible with Z/Eves, such as proving that key properties hold in the specification, so the specification can be checked for other types of correctness.

The next chapter will take these formal specifications and describe the various synthesis activities possible for generation of implementation components.

8

RAPTOR Synthesis

Given the formal notations and structure of the RAPTOR specifications presented in the previous chapter, a key solution principle is to synthesize as much of the implementation as possible from the specification. This chapter outlines the requirements of the application and implementation architecture in order to enable generation and integration of synthesized fault-tolerance code, then describes in detail the synthesis of implementation components from the formal specifications.

Figure 16 depicts both the system architecture and a detailed breakout of an application node together to show the implementation components synthesized by the RAPTOR System. The generated code components are shaded and include the following:

- Control System augment (for effecting control with abstract finite-state machines)
- System Specification database (for storing configuration information)
- Sensor code (for sending node-level error detection information from application nodes to the Control System)
- Actuator code (for receiving recovery commands at application nodes from the Control System)
- Event and action messages (for communication between application nodes and the Control System)

This chapter explains in detail each of these generated code components and the process of synthesis. This figure is referred to throughout the chapter when presenting the RAPTOR synthesis activities.

8.1 Application and Implementation Architecture Requirements

The RAPTOR specifications can be utilized in the implementation of a fault-tolerant system given an appropriate application architecture and implementation environment. Before discussing the synthesis of implementation components, it is necessary to outline the implementation architecture into which the generated code will fit. This section

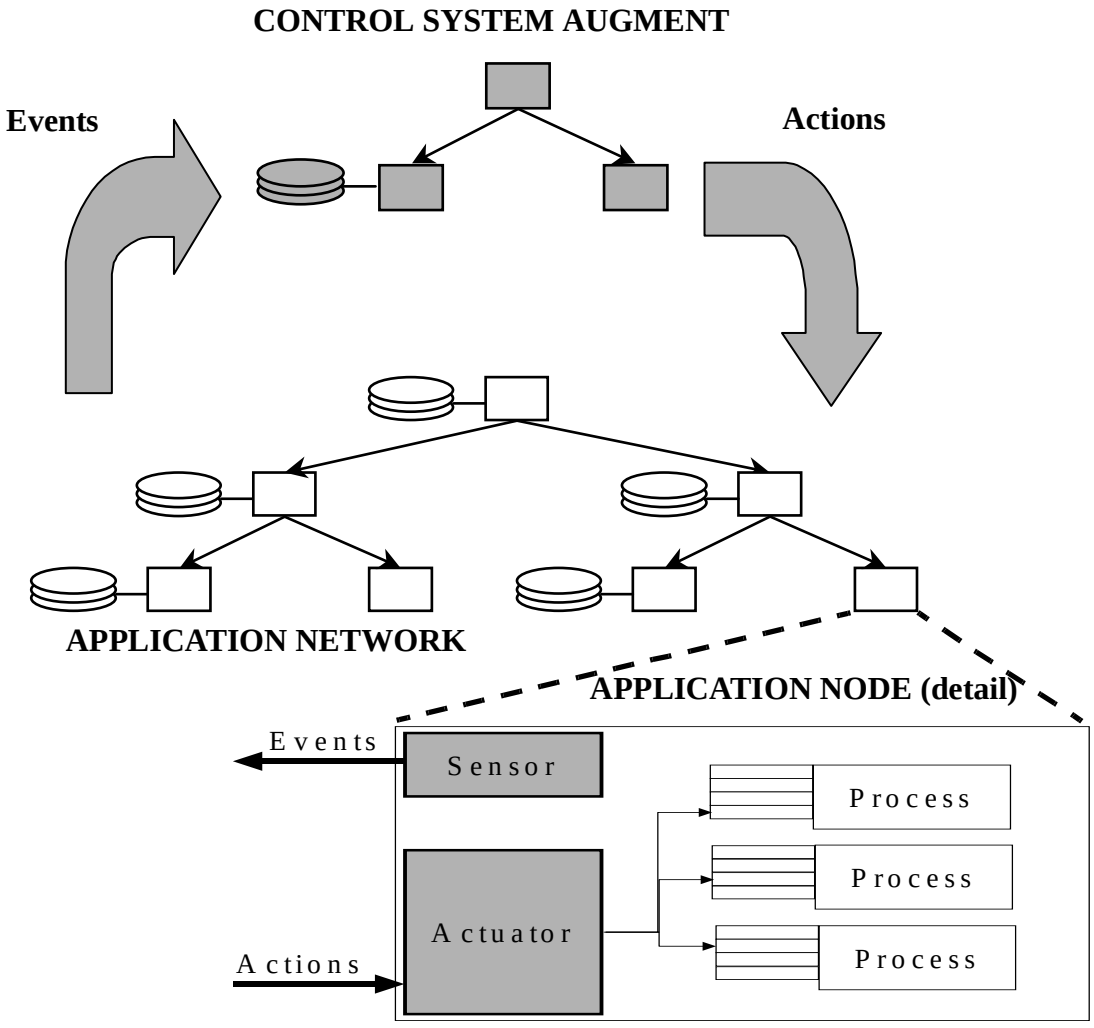

Figure 16: System and node architecture together (synthesized components shaded)

describes the requirements of the application and the overall system structure into which the RAPTOR methodology contributes generated code.

At the application-node level, there are three primary requirements that must be satisfied by the application in order to integrate RAPTOR synthesized code with the node implementation:

1. The application must provide alternate functionality to support recovery.
2. The application must integrate that alternate functionality in a systematic manner to facilitate reconfiguration to alternate functions when necessary.
3. The application must possess low-level error detection capabilities and provide that error detection information for analysis.

While the provision of alternate services required for reconfiguration and recovery is an obvious requirement (#1), the alternate functionality must be structured such that reconfiguration and manipulation of system state is easily accomplished (#2). Regarding

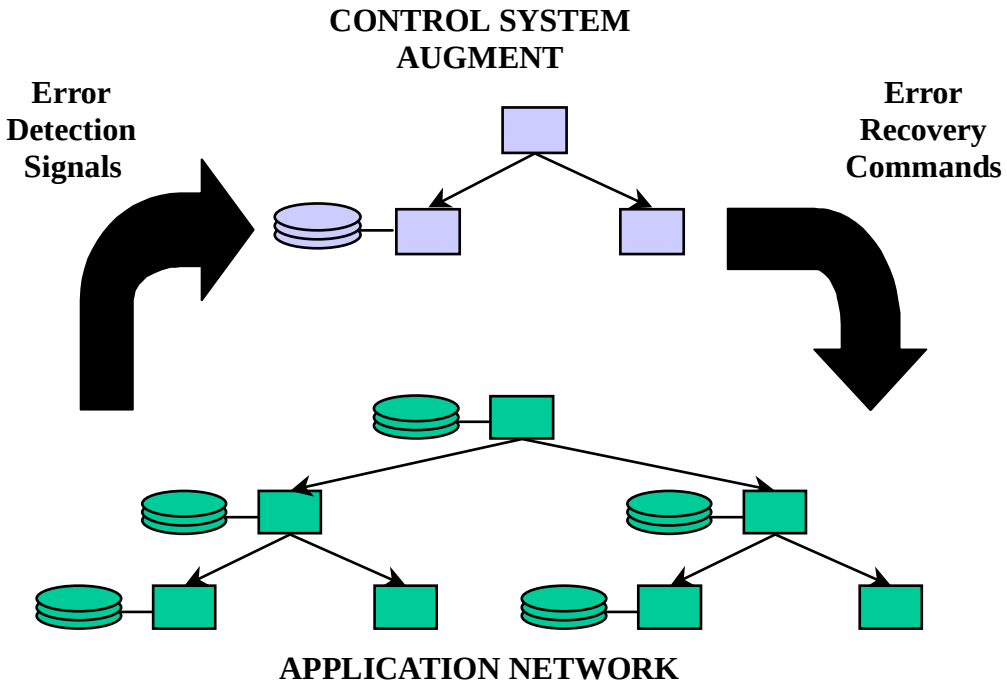

Figure 17: RAPTOR system architecture

low-level error detection, this work does not address those capabilities as mentioned previously. However, the analysis of low-level error detection information is necessary in order to determine the overall system state, therefore there must be an interface by which the application can communicate that information (#3).

At the system level, the primary requirement for incorporating synthesized code is that the application support an augment to the system architecture to perform high-level error detection and coordinate recovery, the Control System (see Figure 17). Many critical systems are augmented with control system architectures to monitor and control processes, utilizing sensors to detect problem conditions and actuators to effect responses to those conditions. Research into control system architectures for critical information systems is an issue that has been explored separately [75]. The architecture of a RAPTOR Control System is not so much an issue, so long as the requirement of a control system augment is satisfied.

8.2 Synthesis of Implementation

Given a reconfigurable application and a control system augment, the RAPTOR specifications can be utilized for synthesis of portions of the implementation that can integrated into the application. The generation of the implementation components achieving fault tolerance leverages off of the development of a formal specification, in addition to yielding

the other benefits mentioned previously (in Section 5.1.2). The following subsections discuss the different areas of synthesis.

8.2.1 System Specification Synthesis

Because relevant characteristics are described in the System Specification using an object-oriented notation associated with an object-oriented database, the system definition is in a form usable by the application immediately. The POET object-oriented database provides a preprocessor to generate database schemas for the database definition automatically [59]. Then at run-time, the database can be instantiated and populated for the given system. (Objects can be instantiated and populated in the database based on a topology specification and additional configuration information.) Finally, during system operation the populated database can be utilized by the Control System to monitor and analyze the state of the system. (Use of the database at run-time though requires that the object-oriented database provide some assurance of adequate performance and achieve any distributed access requirements dictated by the Control System.)

The synthesized database for System Specification information is depicted in Figure 16 by the shaded database symbol (three disks) attached to the Control System.

In the banking system example from the previous chapter, the System Specification definition for a money-center bank (shown in Section 7.2.1) was utilized to define a database schema for that object. Then during system operation, a database object was instantiated for each money-center bank in the system, and the populated database was accessed by Control System nodes to help monitor system state.

8.2.2 Error Detection and Recovery Specifications Synthesis

The RAPTOR specifications describing high-level error detection and error recovery are utilized to generate code by processing them with the Fault Tolerance Translator. This subsection describes the translator, then the generated code components.

Fault Tolerance Translator

The Fault Tolerance Translator was constructed to synthesize code components from the fault-tolerance specifications. The translator accepts as input the Z specifications for error detection and recovery, and produces as output C++ code for various fault-tolerance components. More specifically, the Fault Tolerance Translator processes the LaTeX Z output of the Z/Eves theorem prover from the Zeus system, after the specifications have been type-checked by Z/Eves. The grammar for the translator consists of approximately 90 production rules and 40 tokens (the grammar can be found in Appendix C).

It is important to reiterate that the Fault Tolerance Translator recognizes only a subset of Z: the Fault Tolerance Translator does not solve the general refinement problem. The Fault Tolerance Translator also requires the Z specification to have a particular structure in order for the error detection and recovery descriptions to be recognized. Given an appro-

```
class MoneyCenterBankFSM
{

public: // Interface
   MoneyCenterBankFSM();
   virtual ~MoneyCenterBankFSM();
   int ReceiveEvent(SystemEvents input_event, SystemInput input,
      cset<OutputMessage> *messages, cset<OutputEvent> *events,
      int Time);

   ColorAssignments DisplayColor;

protected: // Data
   unsigned int MyMcb;
   cset<unsigned int> MyBbs;
   cset<unsigned int> HostBbs;
   unsigned int MyFrb;
   McbNodeStates NodesDown;
   bool DbDown;
   bool Alert;
   McbBackupFrbStates BackupFrb;
   cset<unsigned int> BbsUp;
   cset<unsigned int> BbsDown;
   cset<unsigned int> BbsIdAlarms;
   cset<unsigned int> BbsAlarmTimes;
   McbBbNodeStates BbNodeStates;
   McbBbIdStates BbIdStates;
   bool BbCoordinatedAttack;

public: // init functions
   void InitMoneyCenterBankFSM();
   void InitMcbMyMcb(SystemInput input);
   void InitMcbNewBb(SystemInput input);
   void InitMcbNewHostBb(SystemInput input);
   void InitMcbMyFrb(SystemInput input);
};
```

Figure 18: Generated money-center bank FSM class definition

priately structured Z specification, the translator can synthesize fault-tolerance code, as described in the next subsection.

Generated Code Components

The Fault Tolerance Translator generates a C++ class for each type of abstract finite-state machine described in the specification. In addition, a header file with definitions of system events, state enumerations, and application messages is synthesized from the specification files. These abstract finite-state machines and system definitions are key components in

```
int MoneyCenterBankFSM::ReceiveEvent(SystemEvents input_event,
SystemInput input, OutputMessageCSet *messages, OutputEventCSet
*events, int Time)
{
   OutputMessage *msg;
   OutputEvent *event;

   switch(input_event)
   {
   case LocalIdAlarmOn:
      {
         Alert = True;
         event = new OutputEvent;
         event->name = McbIdAlarmOn;
         event->destination = HostBbs;
         event->destination.Append(MyFrb);
         events->Insert(*event);
         msg = new OutputMessage;
         msg->name = MsgMcbRaiseAlert;
         msg->destination.Append(MyMcb);
         messages->Insert(*msg);
         break;
      }
   case LocalIdAlarmOff:
// etc.
```

Figure 19: Generated money-center bank low-level event code

the Control System, thus in Figure 16 the Control System augment is shaded.

The member variables in the class definition for each finite-state machine are derived primarily from the state schema for the finite-state machine. The methods generated for each finite-state machine class are an empty constructor and destructor, initialization functions (from the initialization schemas), and a handler for receiving system events (a function called *ReceiveEvent*).

Figure 18 shows the output of the Fault Tolerance Translator for the money-center bank finite-state machine class. (The money-center bank Z state schema, from which most of this code was generated, was shown in the previous chapter, and can be found again in Appendix C.) In addition to the class methods described above, the member variables are derived directly from the Z state schema: integer ID numbers refer to other banks, including important sets of banks such as branch banks owned by this money-center bank (*cset* is a POET-generated container class). Enumerated type and boolean variables describe state variables for the finite-state machine, including those pertaining to node failure (*McbNodeStates NodesDown*), database failure (*bool DbDown*), and intrusion detection alarms (*bool Alert*). One final state variable, *ColorAssignments DisplayColor*, is generated automatically to facilitate visualization of finite-state machine state.

The *ReceiveEvent* method is the central method in the abstract finite-state machine class, as this is the function that effects transitions between states. The *ReceiveEvent* han-

```
    // switch statement and other high-level event conditions above
    if ((BbCoordinatedAttack == False) &&
        (BbsAlarmTimes.GetNum() >= 10) &&
        (RaptorSeqElem(&BbsAlarmTimes, ((BbsAlarmTimes.GetNum()) - 9))
          >= (Time - 60)))
    {
       BbCoordinatedAttack = True;
       event = new OutputEvent;
       event->name = McbBbCoordinatedAttack;
       event->destination = MyBbs;
       event->destination.Append(MyFrb);
       events->Insert(*event);
    }

//etc.
```

Figure 20: Generated money-center bank high-level event code

dler is synthesized by creating a switch statement from all of the operation schemas for processing low-level events. In addition, the high-level event schemas are used to generate code to check for these conditions at the end of the *ReceiveEvent* handler and to manipulate the state as necessary. The input to this function is a single event from the *SystemEvents* set defined in the specification, any additional node information related to the incoming event, and the global time. The outputs of this function are a set of events to pass to other finite-state machines as a result of this transition and a set of messages that can be sent to the application node(s) manipulated by this finite-state machine in order to respond to the event. This function also manipulates the state according to the relevant operation schema corresponding to the input event.

Figure 19 shows the output of the Fault Tolerance Translator for the *LocalIdAlarmOn* event in the *ReceiveEvent* function for the money-center bank finite-state machine. (The Z operation schema for this particular event was shown in the previous chapter.) The parameters to the function comprise the inputs and outputs described previously. For this event, first the member variable related to the intrusion detection alarm, *Alert*, is set. Then, an event notification for the connected branch bank and Federal Reserve Bank finite-state machines is generated, and a message to the money-center bank application node is generated.

At the end of the *ReceiveEvent* handler is code generated for recognition of and response to high-level events. Figure 20 shows the output of the Fault Tolerance Translator for the $McbConditionSchema10AlarmsIn60Seconds$ high-level event schema (also shown in the previous chapter). The proposition checking whether this event has occurred is translated into the *if* clause. If the conditions are found to be true, then the state variable relating to branch bank coordinated security attack is set and the appropriate event communicated to other finite-state machines.

Finally, at the node-level it is possible to generate shells for sensor and actuator code based on the pre-defined low-level errors and application services. For the synthesized

sensor code, the messages to send the Control System upon occurrence of a low-level failure are defined in the Error Detection Specification. Sensors can be synthesized that generate these messages, and the integration task left with the node application code is tying the message generation to the actual low-level error detection event. For the synthesized actuator code, the messages that the Control System sends to the application are defined by the Error Recovery Specification. Actuators can be synthesized to receive these messages, and the remaining integration task is to call the appropriate node application code upon receipt of the notification message. The sensor and actuator components at the node level in Figure 16 are shaded representing their synthesis, and the arrows for event and action communication are shaded as well.

8.3 Summary

Each of the RAPTOR specifications is utilized to synthesize code components enabling fault tolerance. The System Specification is processed by the POET preprocessor, while the Error Detection and Recovery Specifications are utilized by the Fault Tolerance Translator. The following code components are synthesized from the specifications:

- *Control System augment.* The Fault Tolerance Translator generates code for the abstract finite-state machines of the Control System. This code consists of a finite-state machine class for each node or set of nodes of concern. The abstract finite-state machines are a key components of the Control System, as they perform high-level error detection and effect recovery activity in the application nodes.
- *System Specification database.* The POET object-oriented database processes the System Specification and produces a database for system configuration information. This database can be accessed at run-time by Control System nodes to analyze relevant configuration information during error detection and recovery.
- *Sensor code.* The Fault Tolerance Translator generates code for the application sensor, though this code will need further instrumentation to tie low-level error detection events to the proper sensor event message.
- *Actuator code.* The Fault Tolerance Translator generates code for the application actuator. This code also requires further instrumentation to tie actuator action messages to actual application response code.
- *Event and action messages.* The Fault Tolerance Translator generates the code to process and receive event and action messages between Control System nodes and application elements.

Code for all of the types of information described in the Error Detection and Error Recovery Specifications (described in Chapter 7) can be generated by the Fault Tolerance Translator, including code for the following:

- Finite-state machines
- System events (including both low-level and high-level errors, recovery from errors, and time-based events)
- Communication between finite-state machines
- Messages to application nodes

- Responses to events in terms of changes in state, finite-state machine communication, and application messages

The next chapter expands on the implementation architecture that incorporates these generated code components.

9

RAPTOR Implementation Architecture

The discussion of synthesis in the previous chapter introduced the RAPTOR implementation architecture at a high level. This chapter explores in more depth the implementation architecture at both the node level and system level.

9.1 RAPTOR Node Architecture

The previous chapter outlined the node architecture with respect to the generated code components. This section presents the node architecture in detail, first outlining the basic principles and minimum requirements of the node-level architecture, then discussing implementation details in the current RAPTOR system and possible enhancements.

9.1.1 Basic Principles

The previous chapter presented three requirements that must be satisfied by an application node supporting RAPTOR fault tolerance:

- The provision of alternate functionality (for error recovery)
- The structuring of the application node to facilitate reconfiguration to alternate functionality
- The provision of error detection capabilities and information

These three application requirements basically ensure that an application program supports the low-level capabilities required for performing the fault-tolerance activities of error detection and error recovery.

The three application requirements also point to the basic principles in structuring the node architecture. Given an application node that satisfies those three requirements, the node architecture can be constructed around that application program. There are three minimal required elements in the node architecture to support fault-tolerance activities: (1) sensors, (2) actuators, and (3) reconfigurable processes. Sensors and actuators were

introduced in the previous chapter: these two components provide interfaces for manipulation of the error detection and error recovery capabilities provided by the application, respectively. This subsection will define in detail the capabilities and characteristics of reconfigurable processes.

A reconfigurable process is a specialized type of application process that is used as the building block for nodes in the RAPTOR System. Firstly, a reconfigurable process implements some aspect of required system functionality, just as any application process would. Secondly, a reconfigurable processes satisfies the requirements of application programs cited previously, basically providing low-level error detection and error recovery capabilities that must be manipulated under the appropriate circumstances. The key specialization of a reconfigurable process, though, is that reconfigurable processes support a set of *critical services* required for reconfiguration. These critical services provide the interface by which a reconfigurable process can switch to an alternate service mode in a safe manner.

At a minimum, reconfigurable processes must provide two critical services in order to effect reconfiguration: (1) the ability to stop current process functionality, and (2) the ability to start new process functionality. Stopping current process functionality must be accomplished in such a manner as to preserve the state of the application safely. Achieving this requirement is made simpler by the fact that a reconfigurable process does not have to be stopped at any arbitrary point: a reconfigurable process can provide specific stopping points so long as they are sufficiently frequent as to maintain reasonable response times to reconfiguration requests, as dictated by the application domain. Starting new process functionality requires access to relevant application state for the given reconfigurable process.

The composition of these two critical services, stopping and starting process functionality, provides the capability of switching to an alternate service mode (obviously, by stopping the current function and starting an alternate function). So long as access to relevant application state is provided to a reconfigurable process in whatever its mode of functionality, the reconfigurable process can be manipulated to achieve any form of application service and then reconfigured to provide an alternate service.

To summarize, there are minimally three required elements of a node architecture to enable fault tolerance:

- An interface to communicate error detection information, the sensor
- An interface to enable error recovery activities, the actuator
- A specialized application process that provides alternate modes of functionality to tolerate faults and a critical service interface by which the process can be manipulated, the reconfigurable process

Thus, a critical information system will be a collection of reconfigurable processes, distributed on nodes, that cooperate in the normal way to implement normal application functionality. The basic application is constructed in a standard manner as a collection of processes, each of which is enhanced to support critical services to stop and start various modes of application functionality. Each node is supplemented with a sensor to communicate error detection information and an actuator to accept reconfiguration commands. Upon request for reconfiguration to effect error recovery, the reconfigurable processes are manipulated to stop current functionality and start alternate functionality using the critical service interface.

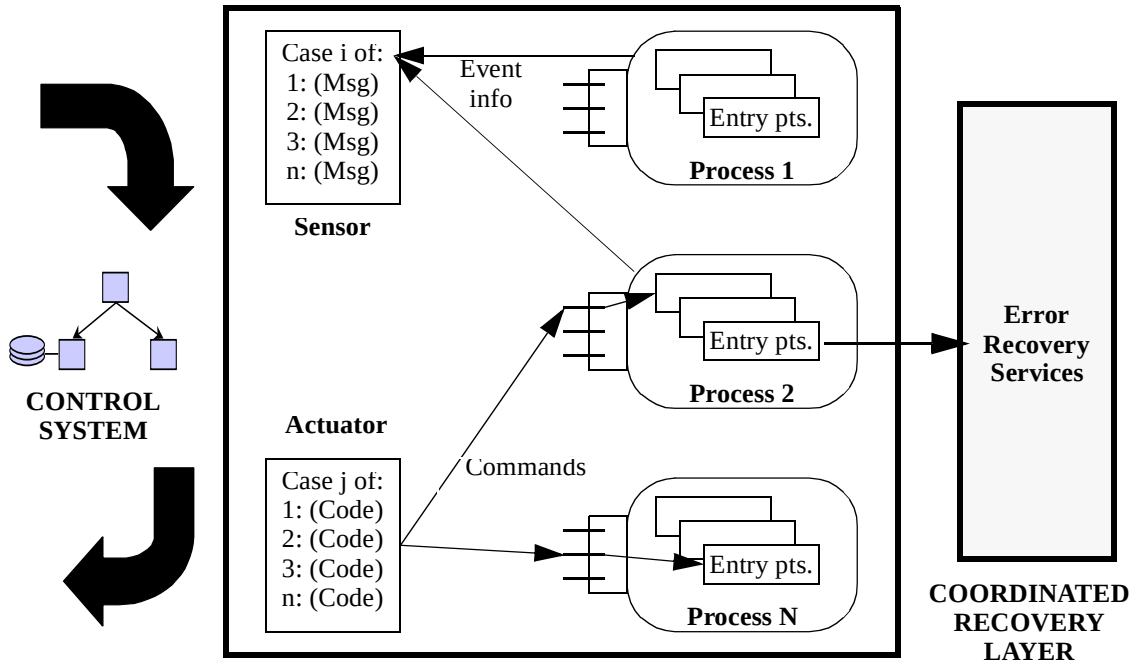

Figure 21: Node architecture (with reconfigurable processes)

9.1.2 RAPTOR Node Implementation

The basic RAPTOR application node has the three primary components just discussed, as pictured in Figure 21: sensors, actuators, and reconfigurable processes. In addition to these node-level components, two system-level entities are pictured: the Control System and the Coordinated Recovery Layer. Sensors, actuators, and the Control System were introduced in the previous chapter, as they are generated by the Fault Tolerance Translator (Section 8.2.2); this subsection introduces the Coordinated Recovery Layer (a detailed description is provided later in this chapter). This subsection discusses implementation issues with respect to sensors, actuators, and reconfigurable processes.

The code generated for a sensor consists of a set of messages that must be sent to the Control System, communicating the occurrence of various low-level events or failures. While the message generation sequences for the low-level events of interest can be generated by the Fault Tolerance Translator, these messages must be tied to the actual detection of errors in the application. Communication must occur between the application code, i.e., reconfigurable processes, and sensor in order to notify the sensor of an error detection event. This communication will depend upon the relationship between the sensor and reconfigurable processes: if the sensor is a separate process, then message passing must take place between the reconfigurable processes. However, it is possible to integrate the sensor into a reconfigurable process; in this case a function call will suffice for communication.

Similar issues arise in the implementation of an actuator. The Fault Tolerance Translator generates actuator code corresponding to the messages that will be received from the Control System to initiate error recovery. Upon receiving a message, the actuator then must make the appropriate calls to the reconfigurable process(es) in order to stop and start modes of functionality. The actuator requires a message-passing interface to receive communication from the Control System. Still, the actuator can either be integrated into a reconfigurable process or be its own separate process; dependent on this design decision, communication with reconfigurable processes will then take place through function calls or message passing, respectively.

The implementation of a reconfigurable process must achieve the requirements for error detection and error recovery already outlined. While these specialized processes can reconfigure themselves when prompted, some recovery activities must be coordinated across multiple application nodes and processes. Coordination is required when several reconfigurable processes must switch to their alternate service modes at the same time, for example. This requires support external to the reconfigurable processes, which is provided in the form of a system-level construct called the Coordinated Recovery Layer. The Coordinated Recovery Layer will be discussed in greater detail in the next section, but it is important to note that the reconfigurable processes can access coordination services during reconfiguration and recovery.

9.1.3 Enhanced Requirements

The node architecture in the current version of RAPTOR supports the fault-tolerance activities of high-level error detection and error recovery as described in this chapter. However, there could be enhanced requirements that require additional architectural support at the node level.

One such enhanced requirement, for example, is bounded response time to reconfiguration requests. In the current RAPTOR node architecture, assurance cannot be provided about response time to a reconfiguration request because no guarantees can be made concerning the scheduling of processes or the processing of messages. If bounds on response time or performance of the reconfiguration mechanism are concerns, the node architecture must be enhanced to provide these guarantees.

There are a number of strategies and mechanisms that could be employed for providing bounded response times in the current node architecture. Firstly, the actuator would have to be constructed as a high-priority process on the node. One mechanism that could be used is an interrupt-driven actuator implementation. Messages received for the actuator on a node could be tied to an interrupt, thus ensuring that the actuator process is scheduled immediately upon receipt of a reconfiguration message from the Control System.

Then, the reconfigurable processes would have to be constructed such that they process critical service messages from the actuator in bounded time. This would involve either another interrupt mechanism or polling on the part of a reconfigurable process at a guaranteed frequency. Both implementation strategies have issues of concern. With the interrupt mechanism, the reconfigurable process would now have to be constructed to accept critical service requests at any point: stopping a running process at any point is a

complex undertaking when considering the consistency of the state being manipulated. With the polling mechanism, a reconfigurable process would have to be scheduled frequently enough such that it could poll for critical service requests from the actuator; in addition, the frequency of polling would limit the amount of actual application processing that could be accomplished before polling again for critical service requests. Both approaches would require operating system support.

9.2 RAPTOR System Architecture

The previous chapter provided an overview of the RAPTOR system architecture with respect to generated code components. This section discusses the system architecture in greater detail, first outlining the basic principles and minimum requirements for the system-level architecture, then presenting implementation details and possible enhancements.

9.2.1 Basic Principles

The previous chapter pointed out one requirement for support of fault tolerance at the system-level: a control system augment to the application to perform high-level error detection and to initiate error recovery. As mentioned previously, a key component of a control system is a network of communicating finite-state machines that perform analysis and correlation of low-level event and error information in order to detect complex, non-local faults, as well as prompt recovery through generation of application messages.

There is a second requirement at the system-level that was touched upon in the previous subsection: support for coordination between reconfigurable processes. The requirement for coordination between processes is clear; there will often be circumstances when a switch to an alternate mode of functionality must occur in multiple nodes at the same time. For example, the switch from communicating with a primary node to communicating with a backup node must be accomplished at roughly the same time in order for application messages to be sent to the correct destination (primary or backup) and for that destination to be prepared for those messages.

The requirement of support for this coordination at the system level is pragmatic. It is possible that every reconfigurable process could be implemented with functionality to enable coordination with certain other nodes in forseeable recovery circumstances. It would seem more efficient, however, to implement the protocols for coordination once in a system-level construct that would then provide these coordination services to all of the reconfigurable processes, enabling coordination between arbitrary sets of nodes, upon request. In large-scale systems, issues in implementation and distribution of this system-level construct must be considered such that support for coordination does not become a bottleneck.

In summary then, there are two requirements for system-level support of fault tolerance: (1) a control system to perform high-level error detection and initiate error recovery activities, and (2) system-level support for coordination services between reconfiguring nodes.

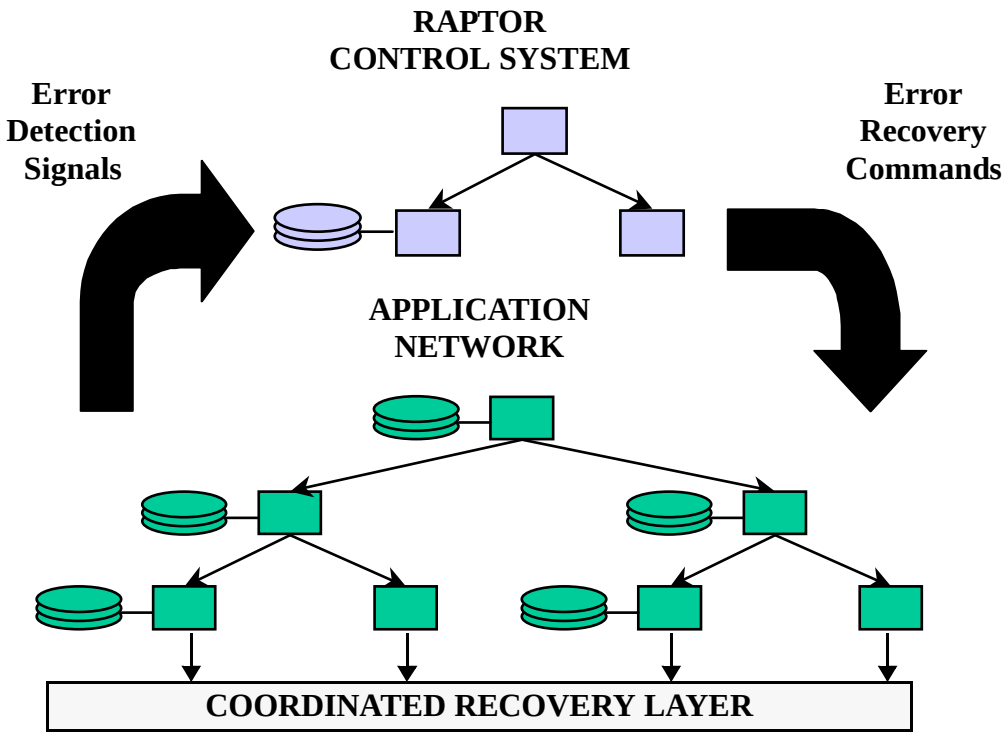

Figure 22: RAPTOR system architecture (with Coordinated Recovery Layer)

9.2.2 RAPTOR System Implementation

The RAPTOR system architecture is pictured in Figure 22, including the two system-level constructs in the RAPTOR implementation architecture: the RAPTOR Control System and the Coordinated Recovery Layer.

RAPTOR Control System

The Control System consists of a set of communicating finite-state machines that perform analysis and correlation of low-level event and error information in order to detect complex, non-local faults. The Fault Tolerance Translator generates code for these finite-state machines. In addition, the Control System can access a database of configuration information, synthesized from the System Specification.

The implementation of a RAPTOR Control System involves construction of nodes that support the activities of synthesized finite-state machines. In particular, the Control System implementation will receive error detection messages from the application node sensors and must pass the relevant information to the appropriate finite-state machine. In addition, the Control System implementation must generate system-specific messages to application node actuators, given abstract instruction to generate messages in the finite-

state machine code. (In other words, the finite-state machines will dictate a message to send and a set of nodes for the destination, but the Control System implementation must translate or adapt this instruction for the particular host and communication mechanism on which the Control System is operating.)

It is important to note that the implementation architecture of the Control System is independent of the network structure of the specified and generated finite-state machines. For reasons of performance and security, the Control System will most likely be constructed to run on dedicated machines, separate from the application network that it monitors (also known as the *controlled* system). The architecture of the Control System can be optimized based on many factors: the Control System can run in a centralized environment or can itself be distributed. The network of finite-state machines will themselves be distributed amongst Control System nodes, presumably optimized for efficient communication between finite-state machines and the application nodes being monitored.

Coordinated Recovery Layer

The Coordinated Recovery Layer provides error recovery services to reconfigurable processes requiring the coordination and control of multiple processes. These error recovery services will be provided in a largely application-independent manner; thus, a common Coordinated Recovery Layer implementation could be used by multiple applications with initial configuration achieved by generation parameters such as the target system topology.

For the current version of RAPTOR, a simple, prototype Coordinated Recovery Layer was implemented to coordinate reconfiguration activities for multiple nodes. The prototype Coordinated Recovery Layer utilizes a partial implementation of the two-phase commit protocol to ensure coordinated commitment to reconfiguration activities between multiple reconfigurable processes. The services of this Coordinated Recovery Layer are implemented as a library available to all application nodes. The Control System dictates a coordinator in the two-phase commit protocol for each recovery activity requiring Coordinated Recovery Layer services. When the Control System initiates recovery activities upon detection of an error of concern, it knows all the participants in the reconfiguration; thus, the Control System can determine a key node in each recovery sequence and select the appropriate coordinator for the commit protocol.

9.2.3 Enhanced Requirements

In terms of the implementation of the system architecture, the prototype Coordinated Recovery Layer demonstrates feasibility of coordination services for application nodes. Coordinated commitment is one basic service required by reconfiguring nodes, and the two-phase commit protocol is but one implementation alternative for that service. It is certainly possible that different reconfiguration activities will require more sophisticated services than coordinated commitment, or that another commitment protocol would be more appropriate depending on the characteristics of the application domain in question.

One possible enhancement at the system-level is the introduction of a more efficient

communication mechanism. In the current RAPTOR implementation, the Control System communicates directly with nodes requiring reconfiguration. Direct, unicast communication with large numbers of nodes is not the most efficient communications protocol, especially (1) when the number of nodes requiring reconfiguration is very large and (2) if the Control System is not distributed in such a way to make that communication efficient. An intermediate communications paradigm could be introduced to make the distribution of event information and recovery commands between a large number of application nodes and the Control System more efficient. A communication paradigm that shows promise is the publish/subscribe methodology for disseminating information between large numbers of nodes [16]. Significant testing has been done with large numbers of nodes, and it has been shown that data can be distributed in an efficient manner [17]. Using a publish/subscribe system as a transport layer, reconfiguration commands could be published by Control System nodes and subscribed to by actuators, with the underlying publish/subscribe network routing commands efficiently and automatically.

9.3 Summary

The RAPTOR implementation architecture consists of components at both the node level and system level that enable fault tolerance. At the node level, sensors, actuators, and reconfigurable processes provide low-level error detection information and error recovery capabilities in a systematic application structure. At the system level, the Control System performs analysis on error events in order to detect errors of concern and prompt the application to recover or reconfigure appropriately. In addition, the Coordinated Recovery Layer provides coordination services to application processes involved in reconfiguration.

10

Evaluation Approach

This chapter outlines the evaluation methodology and presents the research questions with which the solution approach is evaluated. In addition, the modelling and experimentation system used as part of the evaluation approach is described.

10.1 Evaluation Approach

Ideally, evaluation of the solution approach would occur by running controlled experiments on actual critical information systems. For a number of reasons this approach is quite impossible. Firstly, access to critical information systems is highly restricted. In most cases, these systems operate continuously, around the clock, year round, and cannot be stopped for testing or experimentation; permission for access to particular information systems often resides with different authoritative domains; and finally, the consequences of failure in these systems are so extreme that they will not be changed under anything but the most drastic circumstances (certainly not for experimental research). Secondly, running controlled experiments on systems this complex and this large is infeasible: it is impossible to control all of the variables due to the size and complexity of these systems. Also, it would be prohibitively expensive to conduct enough trials for any single experiment to be statistically valid.

The approach to evaluation must proceed pragmatically. Given that experimentation on actual critical information systems is not possible, representative models of these systems were constructed in two infrastructure application domains. These representative models are derived from domain analysis, focusing on the relevant characteristics of these systems outlined in the chapter on problem description (Chapter 2). The models will be used for experimentation as part of the evaluation approach.

To guide the evaluation of this research, the next section presents research questions relating the solution requirements to the major solution components.

10.2 Key Research Questions

Each aspect of the solution must be evaluated with respect to the solution requirements listed in Chapter 5. The major solution areas and their key components that must be considered during evaluation are the following:

- Specification
  - — System Specification
  - — Error Detection Specification
  - — Error Recovery Specification
- Synthesis
  - — Fault Tolerance Translator
  - — Database synthesis
- Implementation Architecture
  - — Sensor/Actuator implementations
  - — Reconfigurable process implementation
  - — Control System architecture
  - — Coordinated Recovery Layer

Evaluation will consist of a set of key questions relating the major solution area to each of the four solution requirements (presented in Section 5.2):

- Scale
- Heterogeneity
- Complexity
- Performance

As a baseline, evaluation can be thought of as a matrix of questions relating each solution requirement to each solution component, where applicable:

> *Can the given solution component of the RAPTOR System accommodate each solution requirement of critical information systems?*

The following is a more detailed set of research questions that will be used to evaluate this research.

Scale

- *Specification.* Do the specification notations and methodology enable description of large-scale systems, errors involving large numbers of nodes, and error recovery activities involving large numbers of nodes?
- *Synthesis.* Can the Fault Tolerance Translator generate sensor and actuator code for a large-scale system and its associated control system? Can the database be synthesized to accommodate a large-scale system?
- *Implementation Architecture.* Can the Control System monitor and effect control for a large number of nodes? Can the Coordinated Recovery Layer accomplish coordination between a large, distributed set of nodes?

Heterogeneity

- *Specification.* Do the specification notations and methodology enable description of heterogeneous nodes, the faults involving heterogeneous nodes, and the different types of recovery response possible in those nodes?
- *Synthesis.* Can the Fault Tolerance Translator generate sensors and actuators for heterogeneous nodes, as well as a control system to monitor and effect control for those nodes? Can the synthesized database accommodate the descriptions of heterogeneous nodes?
- *Implementation Architecture.* Does the reconfigurable process architecture provide a feasible structuring mechanism for the different types of heterogeneous node? Can the Control System monitor and effect control over heterogeneous nodes? Can the Coordinated Recovery Layer provide coordination services for those heterogeneous node types?

Complexity

- *Specification.* Do the specification notations and methodology enable description of the complex functionality and architecture of critical information systems, as well as the complex faults of concern and recovery responses?
- *Synthesis.* Can the Fault Tolerance Translator generate sensors and actuators for a complex application, as well as its associated control system?
- *Implementation Architecture.* Can the Control System monitor and effect control in a complex system? Can the Coordinated Recovery Layer support coordination activities for complicated recovery responses?

Performance

- *Specification.* Do the specification notations and methodology enable description of the performance-related aspects of the system, faults, and responses?
- *Synthesis.* Can the Fault Tolerance Translator generate code that supports the performance-related aspects of a critical information system and its associated control system?
- *Implementation Architecture.* Can reconfigurable process architecture support and achieve application performance requirements during recovery and reconfiguration? Can the Control System monitor and effect control in an efficient manner? Can the Coordinated Recovery Layer provide coordination services within performance bounds?

These key research questions will attempt to demonstrate, as a baseline, the feasibility of the RAPTOR System in achieving fault tolerance in critical information systems. While one cannot *prove* that the RAPTOR System will work for *all* critical infrastructure domains, this work attempts to show that the RAPTOR approach is applicable to critical information systems by addressing the aforementioned research questions using three evaluation methods: (1) rigorous argument, (2) analysis, and (3) experimentation.

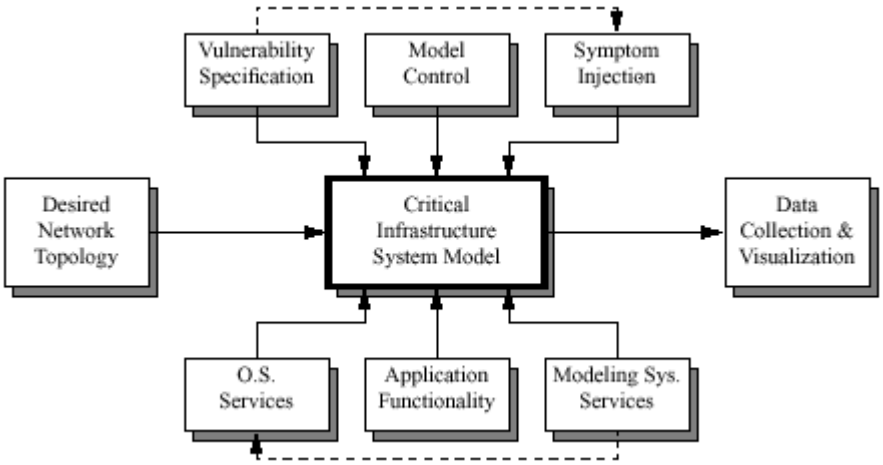

Figure 23: RAPTOR Simulator environment

The next section describes the system used for experimentation with critical information system models.

10.3 RAPTOR Simulator

The Survivability Research Group in the Department of Computer Science at the University of Virginia has developed modelling and experimentation systems for research on critical information systems [74]. The current modelling system is called the RAPTOR Simulator [69]. The RAPTOR Simulator provides a general network simulator with additional tools important for experimentation on critical infrastructure information systems. These tools allow the user to model network faults such as security attacks, system failures, and a variety of other user-defined events [69].

The RAPTOR Simulator enables rapid construction of large, distributed models of critical information systems by providing the following capabilities:

- A general message-passing infrastructure
- A mechanism for description and instantiation of arbitrary information system topologies
- A system for vulnerability definition and symptom injection
- A virtual time mechanism to enable measurement various phenomena

Figure 23 shows an overview of the RAPTOR Simulator environment. A RAPTOR model is defined by describing the desired topology and implementing the desired application functionality. Based on the topology, the model is created using services from the modelling system's support libraries and using the application software provided by the model builder. Faults to which the model should be subject are defined and controlled by a

user-defined fault script. During the execution of a model, these faults can be injected into the model to indicate any event of interest to the user. Events might include security penetrations, hardware failures, etc. Any data of interest to the user can be collected and made available to a separate process (possibly on a remote computer) for display and analysis. Finally, since multiple independent models can be defined from separate topology specifications, complex systems of interdependent critical networks can be modelled [69].

The current version of the RAPTOR Simulator runs on the Windows 2000 platform and is capable of running tens of thousands of Windows threads for modelling purposes [40]. For this work, a single Windows thread is used to model each node, but the assignment of nodes to threads in the RAPTOR Simulator is arbitrary and user-defined. Threads are capable of simulating arbitrary application functionality and can be instrumented to respond appropriately to fault injection and experimental measurement.

It is important to note that while these models simulate application functionality, the models themselves are *actual concurrent systems*, with all of the complexity inherent in concurrent systems programming. Because every node is simulated by a separate thread in these models, in a 10,000-node model there are actually *10,000* threads running, each with its own memory and thread of control. All of the communication and synchronization issues inherent to concurrent applications programming are present in these models, and, very importantly, the issue of scale is *not* abstracted away. Thus, while evaluation of the solution approach on a model is not necessarily ideal, these models are fairly accurate representations of critical information systems in certain key dimensions, and thus suitable for evaluation.

Two example critical information systems will be examined in detail to ensure that the solution approach addresses the key concerns in two important application domains: (1) banking and finance, and (2) the electric power system. Models of the financial payments system were constructed after meetings with information technology managers at the Federal Reserve Automation Services in Richmond, VA. In addition, models of the electric power grid were constructed after meetings with information technology managers at Virginia Power. (Though the final form of neither model was evaluated by domain experts after construction.)

11

Experiments and Analysis

This chapter describes in detail the systems built in two application domains using the RAPTOR approach to fault tolerance. In addition, the experiments and analysis performed using those systems are presented.

11.1 Financial Payments System Experiments

As part of evaluation of the solution approach, models of banking systems were built. This section presents a 10,000-node financial payments system model, its specifications and implementation, and experimental data from that system.

11.1.1 Scenario Description

The model of a financial payments network was constructed in order to demonstrate and evaluate the capabilities of the RAPTOR System on a system resembling the United States financial payments system. The banking network model consists of three types of bank nodes, as described in Table 4. This model was constructed to be roughly the size of the U.S. financial payments network in order to demonstrate the RAPTOR System on a realistically-sized, large-scale critical information system; as such the application model consists of approximately 10,000 nodes.

Given this application domain and structure, a control system augment to perform high-level error detection and error recovery was designed. The control system consists of a three-level hierarchy of finite-state machines corresponding to different levels of control: system level, regional level, and local level. The system-level finite-state machine controls the Federal Reserve Bank and its backups and communicates with the regional finite-state machines. The regional finite-state machines correspond to the administrative hierarchy of the banking network and each controls a money-center bank, while being connected to the system-level finite-state machine and local finite-state machines beneath

it. The local finite-state machines monitor thirty-three branch banks each, so that three local finite-state machines are connected to each regional finite-state machine.

Table 4: Financial payments system model description

Bank Type	Number of Nodes	Description
Federal Reserve Bank	1 primary, 2 backup	Top level of the hierarchy; route batches of check requests and responses between money-center banks and maintain overall balances for those money-center banks.
Money-center banks	100	Second, middle level of the hierarchy; batch and relay requests for check service using the Federal Reserve Bank, route requests for check service to branch banks, and maintain balances for branch banks beneath them in the hierarchy.
Branch banks	9,900	Lowest level of the hierarchy; store and maintain customer account balances, accept checks for deposit, and process requests for check service from money-center banks.

With the given banking application and control system in mind, a set of faults were defined at each node and level of the abstraction hierarchy. For each bank node, the events corresponding to local faults included the following: intrusion detection alarm, database failure, power failure, and full node failure. More complex, high-level faults of concern were also defined using combinations of these local faults, including coordinated security attacks and common-mode software failures.

Finally, a set of application responses were defined for each of the prescribed faults. Corresponding recovery events were defined for each local fault, some using masking responses and others non-masking responses. Similarly, error recovery responses were prescribed for the high-level faults of concern.

In practice, a systems engineer familiar with the application domain would determine the faults of concern and appropriate responses for those faults. For purposes of experimentation, reasonable faults and responses were defined based on domain analysis in order to prove the feasibility of error detection and error recovery.

11.1.2 Scenario Specification

The RAPTOR specifications for the banking model are presented in their entirety in Appendix C, Section C.2.

System Specification

The System Specification described each bank type as a class in the POET database. First a base class for all banks was defined, including member variables for bank name and unique bank ID number (to be used as a searchable index). Then the derived classes for Federal Reserve, money-center, and branch banks were defined. Each bank contained references to related banks, such as parent and sets of children banks. Finally, additional node characteristics such as hardware platform, operating system, and power provider (company) were included in the database definitions.

Error Detection and Recovery Specifications

The Error Detection and Error Recovery Specifications written in Z consisted of five files, including three files to model the abstract finite-state machines at the three levels of the banking hierarchy (*FederalReserveBankFSM*, *MoneyCenterBankFSM*, *BranchBankFSM*). Each finite-state machine specification consisted of a state schema, initialization schemas, and a set of operation schemas to specify the transitions taken in the finite-state machines on occurrence of faults. The *BranchBankFSM* specification consisted of four initialization schemas, fifteen low-level event schemas, and four high-level event schemas. The *MoneyCenterBankFSM* specification consisted of five initialization schemas, nineteen low-level event schemas, and eight high-level event schemas. Finally, the *FederalReserveBankFSM* specification consisted of four initialization schemas, eighteen low-level event schemas, and five high-level event schemas.

The Error Recovery Specification consisted of a set of application messages defined in Z, corresponding to application reconfigurations provided in response to faults. The branch bank required eight messages to specify its services, the money-center bank required ten messages, and the Federal Reserve bank required ten messages.

11.1.3 Scenario Implementation

Using the RAPTOR specifications, portions of the implementation were generated for the payments system model. The finite-state machines abstractions written in Z were run through the Fault Tolerance Translator to generate code for sensors, actuators, and control system nodes. In addition, the generated file of system definitions was integrated into the application model. The message definitions for application reconfiguration were used to define response code in application nodes. Finally, the POET database was integrated with the RAPTOR modelling system to enable control system nodes to store and access state information pertaining to application nodes at run-time.

A complete model of the banking application and its control system supplement was constructed after integration of the synthesized code. In order to visualize the running system, a user interface was developed to represent the entire system and devised color codes for each node to denote its current state. In addition, the application and control system code were instrumented to communicate node state information to the user interface during experimentation. Finally, to perform experiments with the model implementation, var-

ious fault scripts were defined and run using the fault injection capabilities of the RAPTOR Simulator. These experiments demonstrate the error detection and error recovery capabilities of the RAPTOR System and are presented in the next subsection.

11.1.4 Experimentation

The first measure of the effectiveness of the RAPTOR System is the effect on overall system performance in terms of service to the end user (the key component of survivability). In the banking model, one measure of overall system performance is check throughput, i.e., the number of checks cleared in a given time period.

Each banking experiment shows the number of transactions (checks processed) over 200,000 time ticks for three different data sets, labelled as the following in each graph:

- *No failures.* The first set of data is the number of transactions when no failures occur in the system; this is the baseline for the experiments.
- *No recovery.* The second set of data is the number of transactions when a set of faults is defined and injected into the system with no specialized recovery enabled.
- *RAPTOR recovery.* The third set of data is the number of transactions when the same set of faults defined previously is injected into a version of the application that was constructed using the RAPTOR System to tolerate faults.

For each experiment, the first 100,000 time ticks are not pictured. The banking system is permitted to approach steady state during that time period. (Steady state in these experiments would be one check transaction being serviced every ten time ticks, which is the rate at which checks are deposited.)

In a system consisting of over 10,000 nodes, there are innumerable combinations of faults that could be injected at various times, such that the total number of possible experiments is infinite. From all the possible experiments, a representative set has been selected to demonstrate the variety of errors that can be detected and some of the recovery responses made possible with the RAPTOR System.

Experiment 1

The first experiment presented involves the failure of a critical node in the system, the Federal Reserve Bank. The failure of this node is considered a non-local fault in the system because the occurrence of this fault would impact all of the money-center bank nodes, and in fact this fault would be catastrophic as most payments processing would cease without the Federal Reserve Bank functionality. This node is so important that a hot-spare backup (actually two) is provided in the banking model, just as in the actual financial payments system. The recovery response in the RAPTOR specification is to switch all processing and have all money-center banks route communications to the backup. Figure 24 shows the data relating to this experiment, where the duration of the fault was from time 125,000 to 175,000. Note that the *RAPTOR recovery* and *No failure* data are almost identical because this fault is masked. While the response to this fault is relatively simple because the system possesses the redundancy to mask the fault, in most cases redundancy

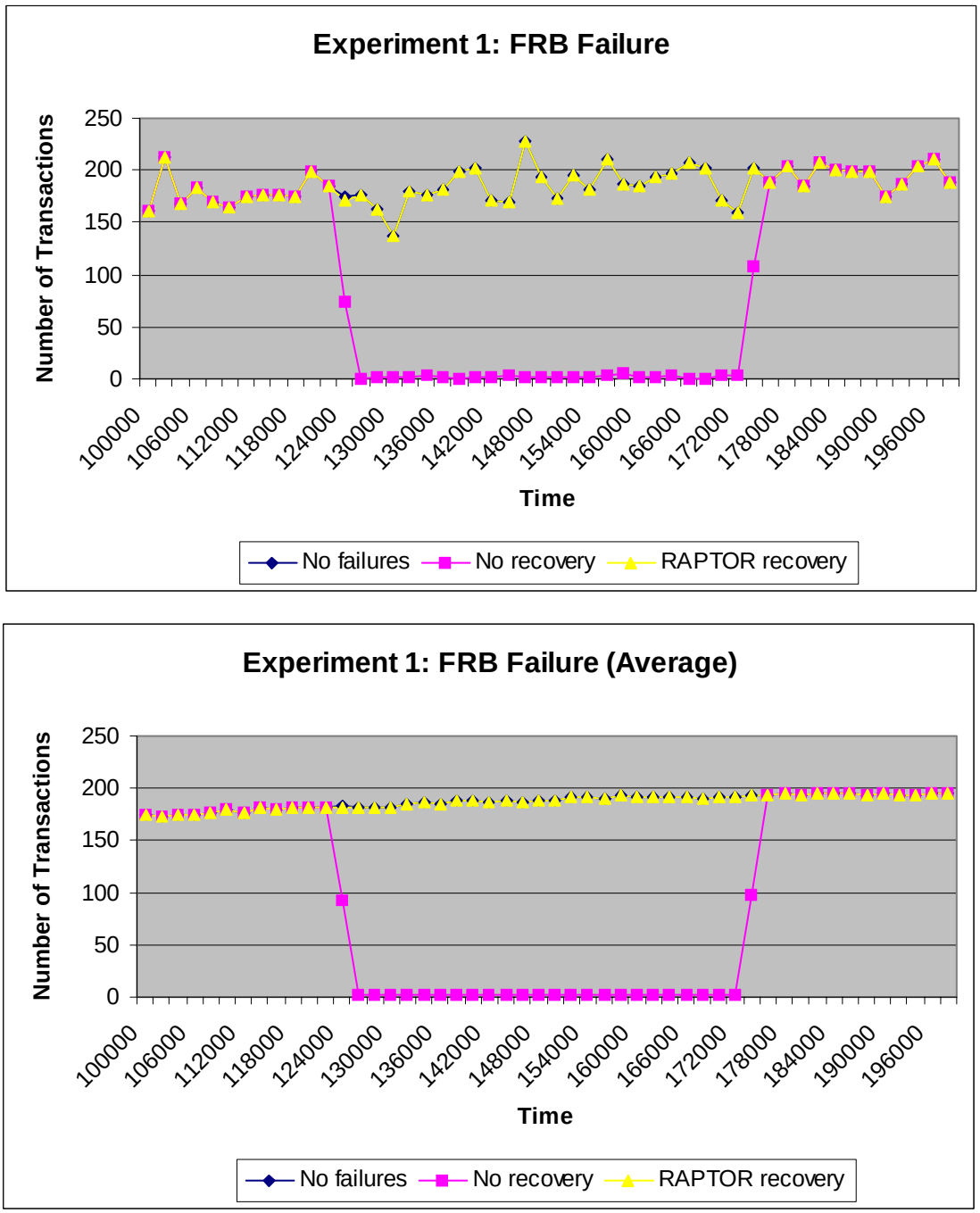

Figure 24: Experiment 1 (Federal Reserve Bank node failure)

is too expensive to provide for non-local faults (as will be seen in the next three experiments). This experiment shows the ability of the RAPTOR System to accommodate masking fault tolerance though.

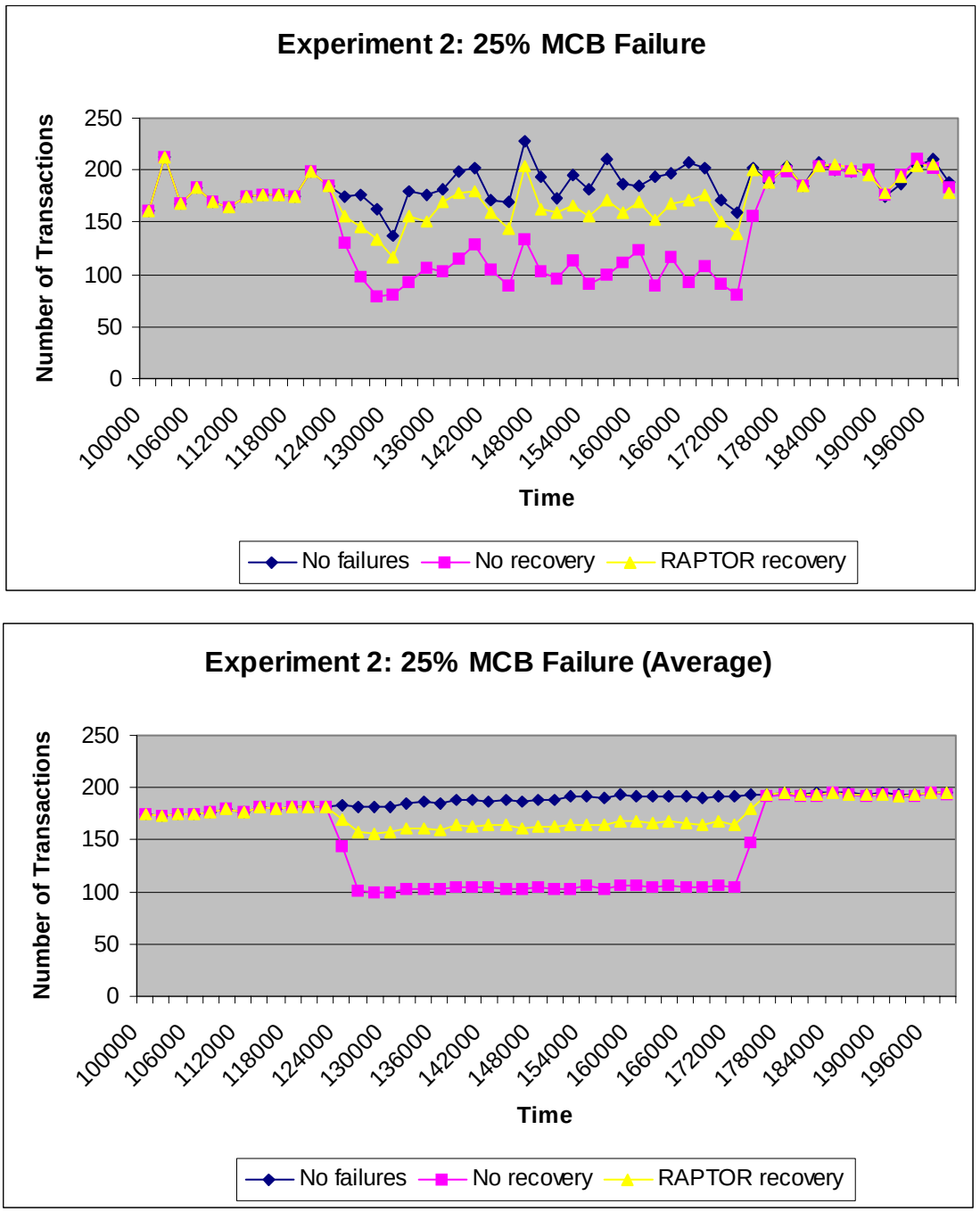

Figure 25: Experiment 2 (25% Money-center bank node failures)

Experiment 2

The second experiment presented involves the failure of 25% of the money-center banks

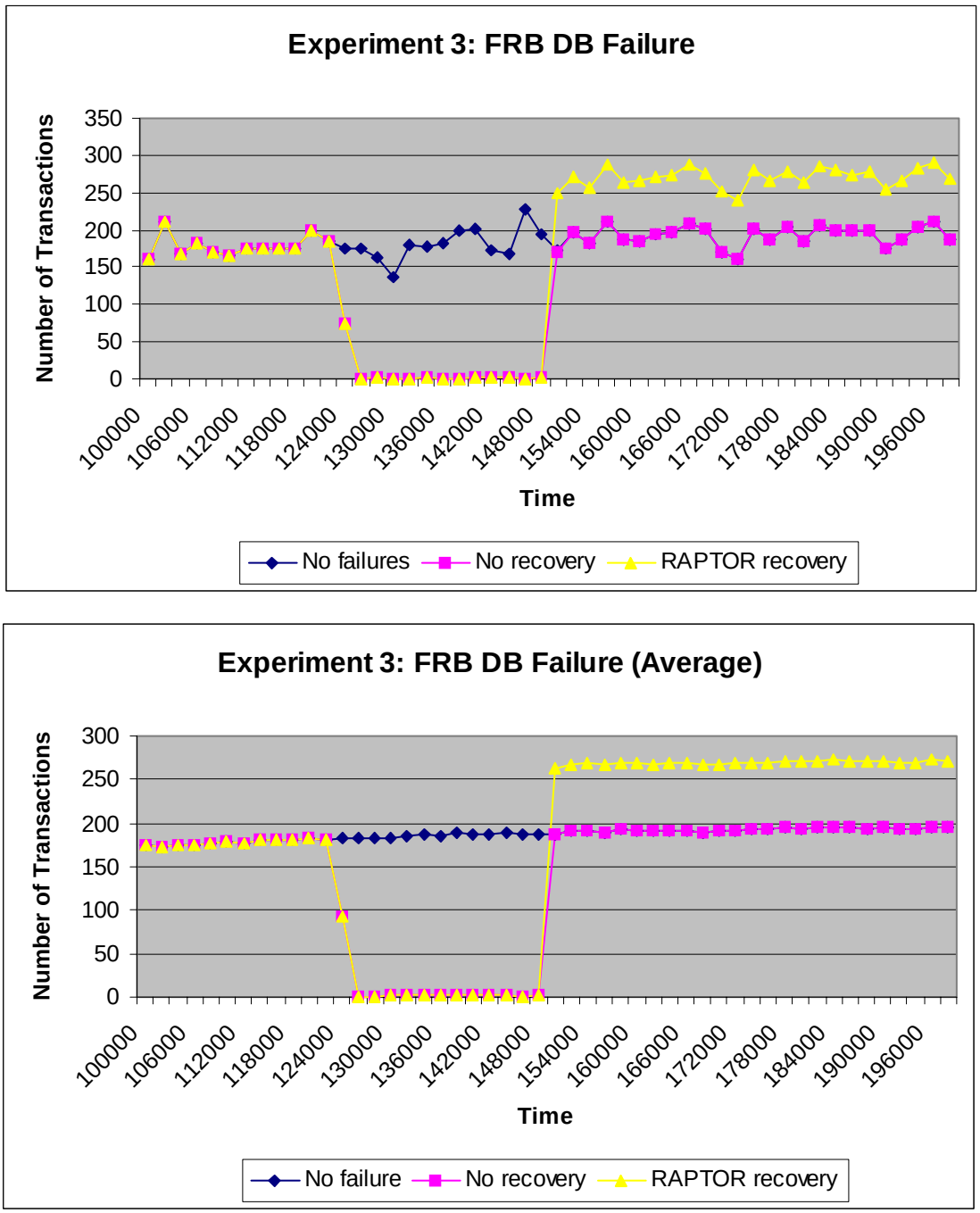

Figure 26: Experiment 3 (Federal Reserve Bank database failure)

in the system simultaneously. While the failure of one money-center bank is a non-local fault because it affects all of the branch banks beneath it, the failure of 25% of the money-center banks would have a significant impact on the level of service provided at the national level. Such an event could occur as a result of a common-mode or cascading soft-

ware failure, and the Control System could detect the cause in such circumstances through analysis of configuration information for failed nodes in the database. The recovery response for this fault in the RAPTOR specification is to designate a branch bank to serve as a backup. The branch bank must then terminate all its local services and reconfigure to its alternate functionality. Figure 25 shows the data relating to this experiment, where the duration of the faults was from time 125,000 to 175,000. Note that this is a degraded service mode because branch bank operations were terminated at those nodes serving as a money-center backup, and the branch bank does not have the processing capacity to handle all of the typical money-center bank workload. The level of service to the end user is substantially higher than if no recovery actions were taken.

Experiment 3

The third experiment presented involves a database failure at the Federal Reserve Bank. This is a non-local fault because it effectively halts processing for those money-center banks beneath it requiring services. For this fault, however, a local, non-masking response is to queue the transactions until the database becomes available again. One might imagine circumstances where the duration of a non-local fault is not anticipated to be very long, and in those cases a local response might be appropriate. Figure 26 shows the data relating to this experiment, where the duration of the fault was from time 125,000 to 150,000. After the fault has ended or been repaired, the queued transactions are processed over time, and thus the increased transaction rates after time 150,000.

Experiment 4

The fourth experiment involves intrusion detection alarms at 10% of the branch banks. The occurrence of an intrusion detection alarm at a single branch bank is not a concern to the survivability of the system: the number of customers affected should one branch bank be compromised is relatively small. However, if error detection analysis reveals an abundance of simultaneous intrusion detection alarms, this could be evidence of a coordinated security attack (where the amount and patterns of concern are defined by a systems engineer) and should be treated as a non-local fault. The response in the banking model is to simulate increase of the encryption strength in communications, thus slowing processing slightly. Figure 27 shows the data relating to this experiment, where the duration of the faults was from time 125,000 to 175,000. In this case, the processing rate for no response to the fault (the *No recovery* line) is not affected because an intrusion detection alarm does not inherently inhibit transaction processing. However, another catastrophic failure, such as a widespread denial-of-service attack, could result if the system did not respond to a high percentage of simultaneous intrusion detection alarms; this is shown in the simulation data at time 150,000 where the coordinated security attacks have crippled the banking network.

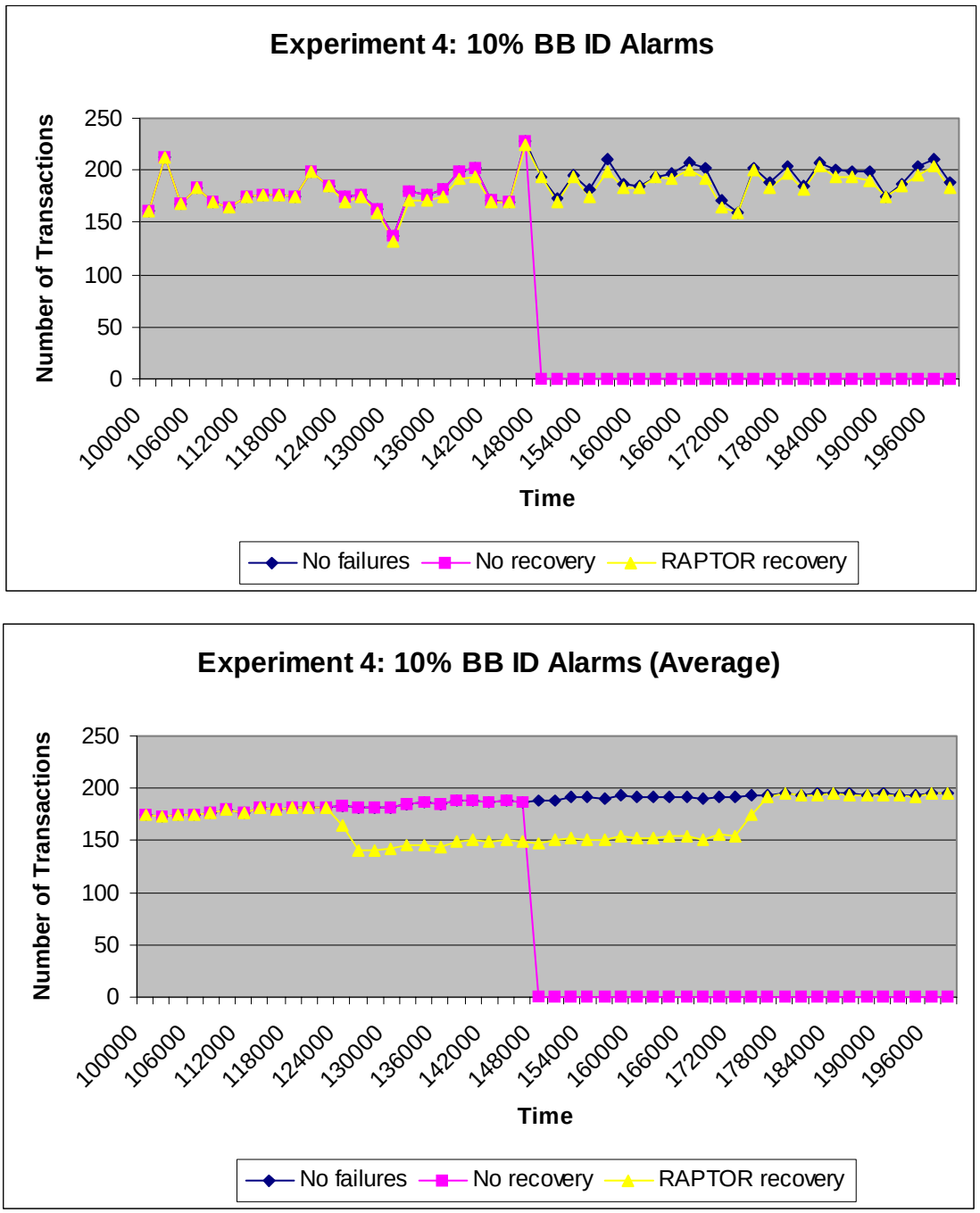

Figure 27: Experiment 4 (10% Branch bank intrusion detection alarms)

Other Experiments

Note that these four experiments are but a small sample of the total number of experiments

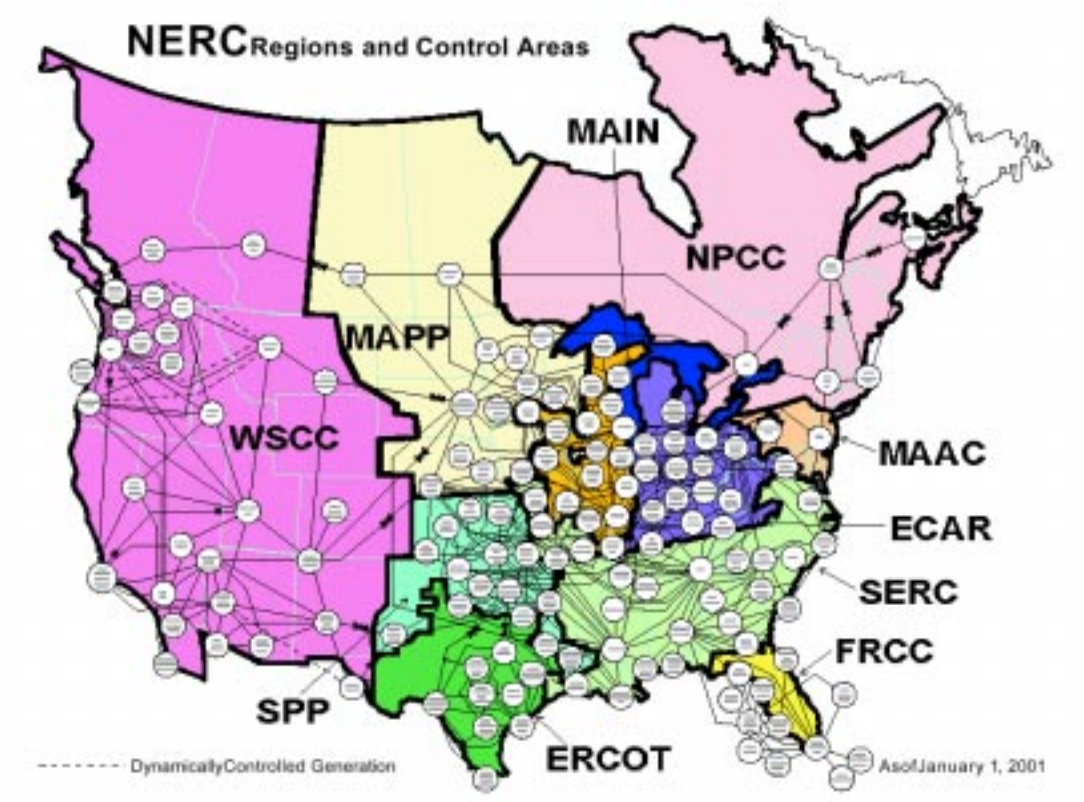

Figure 28: NERC Control Regions and Control Areas [57]

---

that were run and could be run using the banking system model and the RAPTOR Simulator. Other non-local faults that could be detected and responded to include the following:

- Federal Reserve bank intrusion detection alarm(s)
- Money-center bank intrusion detection alarm(s)
- Common-mode software failures involving combinations of bank types
- Database corruption involving combinations of bank types
- Widespread power failures involving combinations of bank types
- Geographic failures (e.g., environmental disaster) involving combinations of bank types

11.2 Electric Power System Experiments

To continue evaluation of the solution approach in a second application domain, models of the electric power system were built. This section presents the latest model, its specifications and implementation, and experimental data from that system.

11.2.1 Scenario Description

The electric power system model focuses primarily on the power generation function and the information systems associated with this function and its reliability.

The upper levels of the model topology correspond to the actual set of entities in the United States electric power grid, as outlined in the NERC operating manual from January 2001 [57]. (See Figure 28 [57].) The model consists of three interconnections: (1) the Eastern Interconnection, (2) the Western Interconnection, and (3) the ERCOT Intercon-

Table 5: Power system model description

Power Node Type	Number of Nodes	Description
Substation	2,574	SCADA system that reports the demand for power to its parent power company.
Generating station	1,287	Information system that controls and reports the supply of power being generated to its parent power company.
Power company	429	Energy Management System (EMS) that accepts data from substation and generator SCADA systems, calculates the balance of power, reports the surplus or deficit to its parent control area, then balances supply and demand with any interchange adjustment accordingly.
Control Area	143	Another EMS that accepts power balances from power companies, calculates and reports the control area balance to its parent control region, then redistributes any adjustment accordingly.
Control Region	10	Information system that accepts power balances from its control areas, calculates and reports the control region balance to its parent interconnection, then redistributes any adjustment accordingly.
Interconnection	3	Information system that accepts power balances from its control regions, calculates its interconnection balance and swaps power with other interconnections according to demand, then redistributes power interchanges amongst its control regions according to demand.

nection. The three interconnections comprise ten control regions, as outlined in Table 2 (see Chapter 2). Finally, distributed amongst the ten control regions are 143 control areas, represented by the small circles in Figure 28.

The lower levels of the model topology are an abstraction of the power grid in terms of power companies, generating stations, and substations. Each control area is responsible for three power companies, and each power company manages three generating facilities and six major substations. The substations in the model signify demand for power, while the generators represent power supply.

It is important to remember that the application model corresponds to the power grid information systems, and not the actual generating and distribution infrastructure (though information system failure does affect generation and distribution). As such, the functionality of each node type in the model is described in Table 5.

Given this application model, a control system to monitor these information system nodes was constructed. The hierarchical control system corresponds roughly to the administrative and regulatory hierarchies in place in the electric power grid. In fact, two overlapping control system hierarchies are modelled: one for general information system monitoring and the other for intrusion and security monitoring. The first, corresponding to the administrative hierarchy, consists of finite-state machines at each power company, control area, and control region. The second, corresponding to the NERC hierarchy for system security issues, consists of finite-state machines at each control area, control region, and the National Infrastructure Protection Center (NIPC).

The low-level faults designed for the system are the following: intrusion detection alarm, database failure, and full node failure. Complex, non-local faults for this model include coordinated security attacks (detected by the NIPC finite-state machine), the loss of significant generating power in a power company or control area, or the failure of key control area or control region nodes. Responses are defined for each fault of concern, including increased generation to compensate for power loss, local balancing functions, and backup processing capacity.

11.2.2 Scenario Specification

The RAPTOR specifications for the electric power model are presented in their entirety in Appendix C, Section C.3.

System Specification

The System Specification described the power company, control area, and control region node types as classes in the POET database. Each class type contained member variables for name, unique ID number (to be used as a searchable index), and references to related power entities, such as parent control area or control region and monitored power companies. Additional characteristics such as hardware platform and operating system were defined in the database.

Error Detection and Recovery Specifications

The Error Detection and Error Recovery Specifications written in Z consisted of six files, including four files to model the finite-state machines at the three levels of the electric power system described above (*ControlRegionFSM*, *ControlAreaFSM*, *PowerCompanyFSM*), plus one additional finite-state machine to model the National Infrastructure Protection Center (*NipcFSM*). Each finite-state machine specification consisted of a state schema, initialization schemas, and a set of operation schemas to specify the transitions taken in the finite-state machines on occurrence of faults. The *PowerCompanyFSM* specification consisted of five initialization schemas, eighteen low-level event schemas, and six high-level event schemas. The *ControlAreaFSM* specification consisted of five initialization schemas, twenty low-level event schemas, and six high-level event schemas. The *ControlRegionFSM* specification consisted of four initialization schemas and eight low-level event schemas (no high-level event schemas). Finally, the *NipcFSM* specification consisted of four initialization schemas, ten low-level event schemas, and six high-level event schemas.

The Error Recovery Specification consisted of a set of application messages defined in Z, corresponding to application reconfigurations provided in response to faults. The power company required ten messages to specify its service modes, the control area required eight messages, and the control region required six messages. The National Infrastructure Protection Center did not have any alternate service modes, as it was strictly an analysis and information processing control system node.

11.2.3 Scenario Implementation

The procedure for generation of the model implementation in the electric power system was exactly as it was for the banking system (presented in Section 11.1.3). The reuse of the Fault Tolerance Translator and POET database system for multiple application domains is a significant advantage of the RAPTOR System.

A complete model of the electric power system and its control system supplement was constructed after integration of the synthesized code. Again, for experimentation various fault scripts were defined and run using the fault injection capabilities of the RAPTOR Simulator. The experiments related to the power model are presented next.

11.2.4 Experimentation

Again, the first measure of the effectiveness of the RAPTOR System is the effect on overall system performance in terms of service to the end user. In the power model, one measure of overall system performance would be the supply of power to customers, represented by substations in the model of the electric power system. Thus, a measure of system performance (or lack of performance in this case) would be the number of power outages experienced by substations.

Each power experiment shows the number of power outages over 1,000 time ticks for

three data sets, the same as described previously (Section 11.1.4):

- *No failures.* The first set of data is the number of power outages when no failures occur in the system; this is the baseline for the experiments and all data points are zero.
- *No recovery.* The second set of data is the number of power outages when a set of faults is defined and injected into the system with no specialized recovery enabled.
- *RAPTOR recovery.* The third set of data is the number of power outages when the same set of faults defined previously is injected into a version of the application that was constructed using the RAPTOR System to tolerate faults.

  Again, a few representative experiments are presented next.

Experiment 5

The first power experiment (fifth overall) presented involves the failure of several critical nodes in the power system, Control Region information systems. The failure of a Control Region node is a non-local fault in the system because the occurrence of this fault would impact all of the connected Control Area nodes. The loss of multiple Control Region nodes, in this experiment half of them, would be quite severe in that power balancing between other Control Regions and the Interconnections would suffer. In the case of this fault, each Control Region node designates a Control Area with redundant processing capacity to serve as a backup. The recovery response in the RAPTOR specification is to switch all processing and have all connected Control Areas route communications to the backup (a masking response). Figure 29 shows the data relating to this experiment, where the duration of the fault was from time 250 to 750. Note that this fault was effectively masked with the exception of the single time interval the system took to react. The remainder of the experiments demonstrate non-masking solutions.

Experiment 6

The sixth experiment presented involves the failure of database services at half of the mid-level nodes in the power system, Control Area nodes. Loss of database capabilities impacts the balancing function at the node. The alternate service provided for this fault is a local response that enables rudimentary balancing between power companies to be performed at the Control Area node. Figure 30 shows the data relating to this experiment, where the duration of the fault was from time 250 to 750.

Experiment 7

The seventh experiment presented involves the failure of generator information systems in two stages, first involving five percent and then ten percent of the generating stations. A single generating station failure is a local fault that will not have a very big impact on the overall system; as such it can be handled at a fairly low level, in this model at the level of the parent power company. The recovery response to the loss of a generating station is to boost power production at the other power companies' generators by some percentage. This does not mask the local fault but it does provide a degraded level of service. When

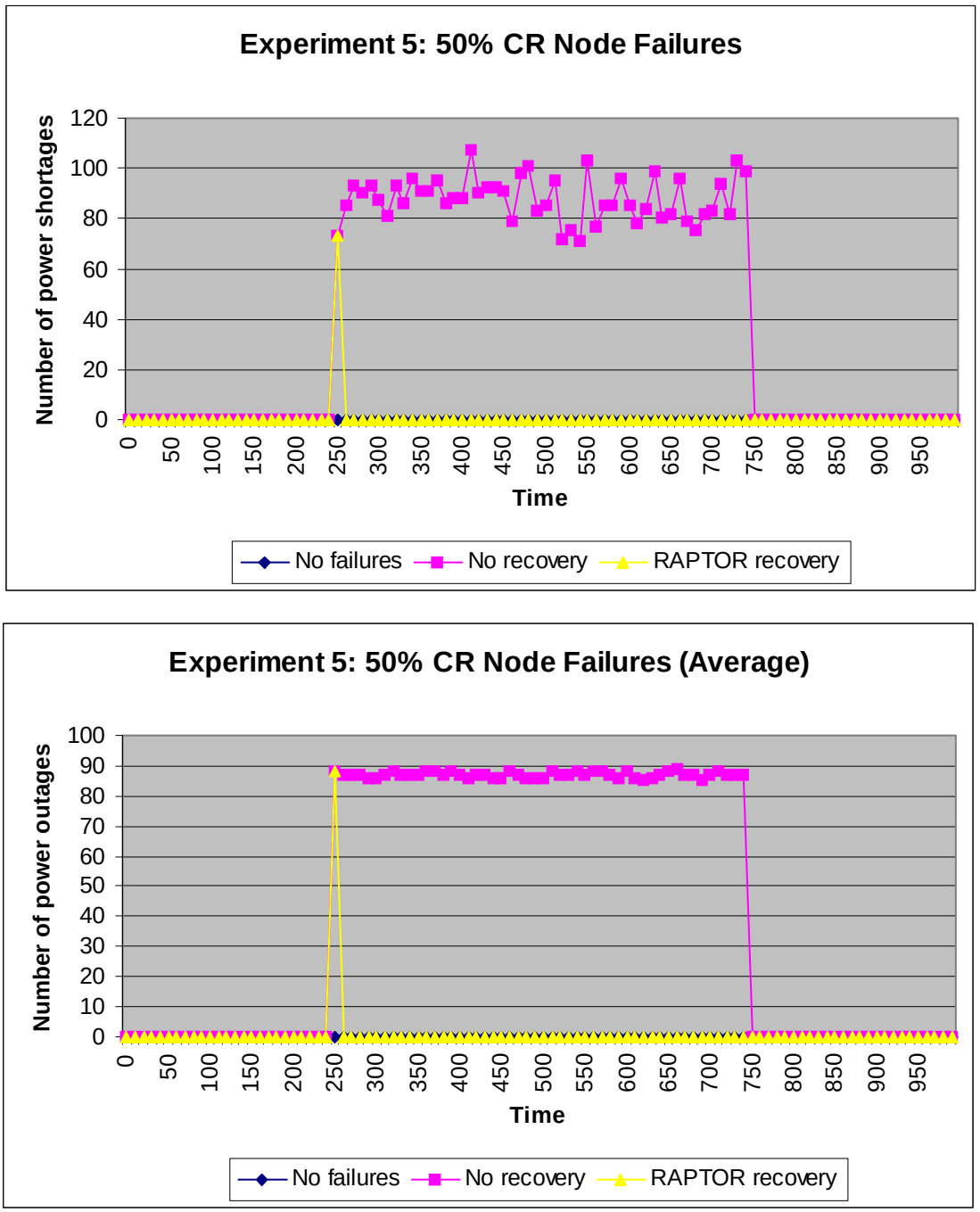

Figure 29: Experiment 5 (50% Control Region node failures)

multiple generators at a power company fail, however, this becomes a concern at a higher regional level. The response to this fault is to boost power production regionally, at the other power companies within the Control Area, in order to compensate more for the fault. Figure 31 shows the data relating to this experiment, where at time 250 five percent of the

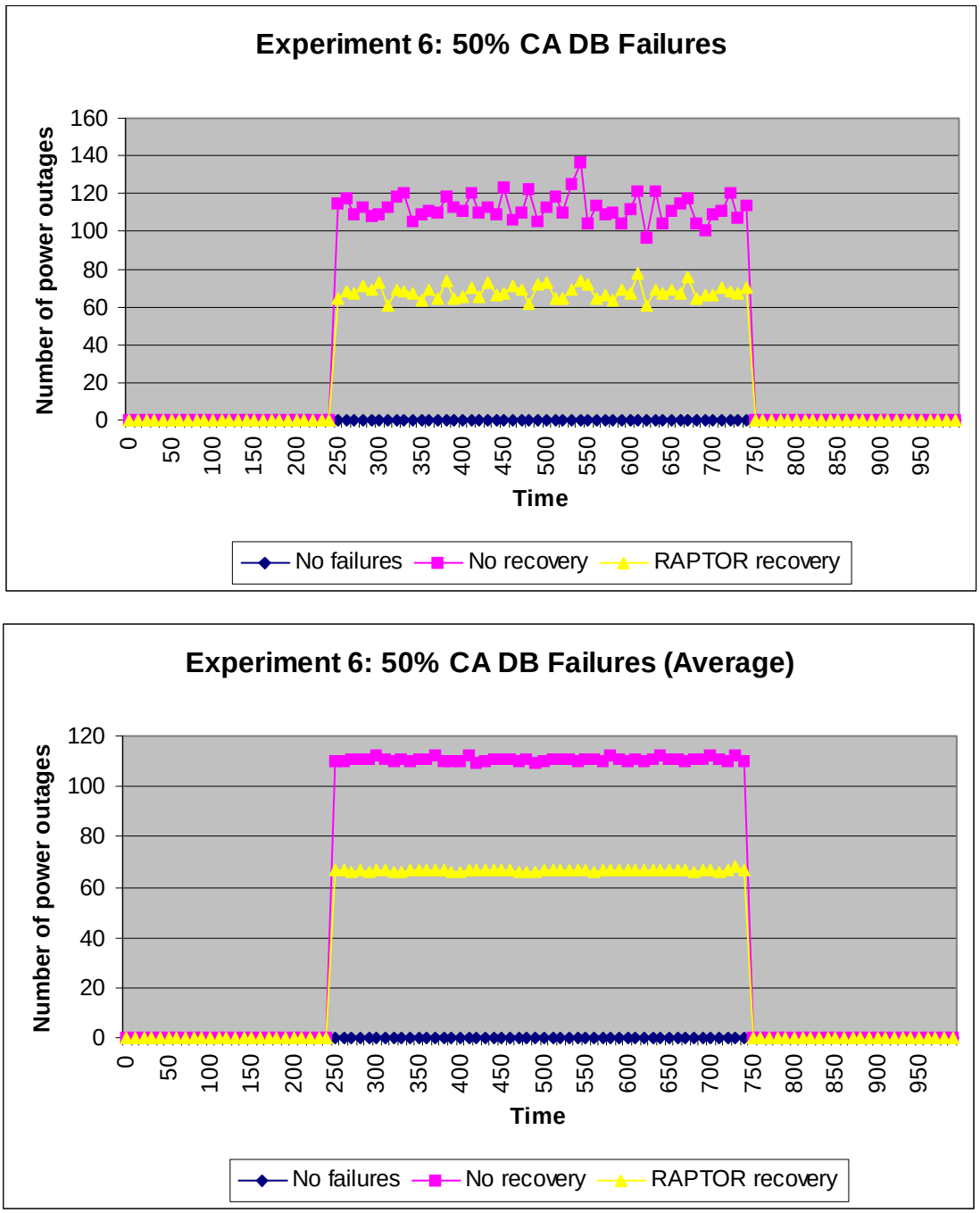

Figure 30: Experiment 6 (50% Control Area database failures)

power companies experienced the loss of a generating station and a local response provided a degraded level of service. At time 500, a cascading fault occurred with the loss of a second generator within each previously affected power company, bringing the loss of generating stations to ten percent. The regional recovery maintained a degraded level of

Figure 31: Experiment 7 (5-10% Generator failures)

service until the faults ceased at time 750.

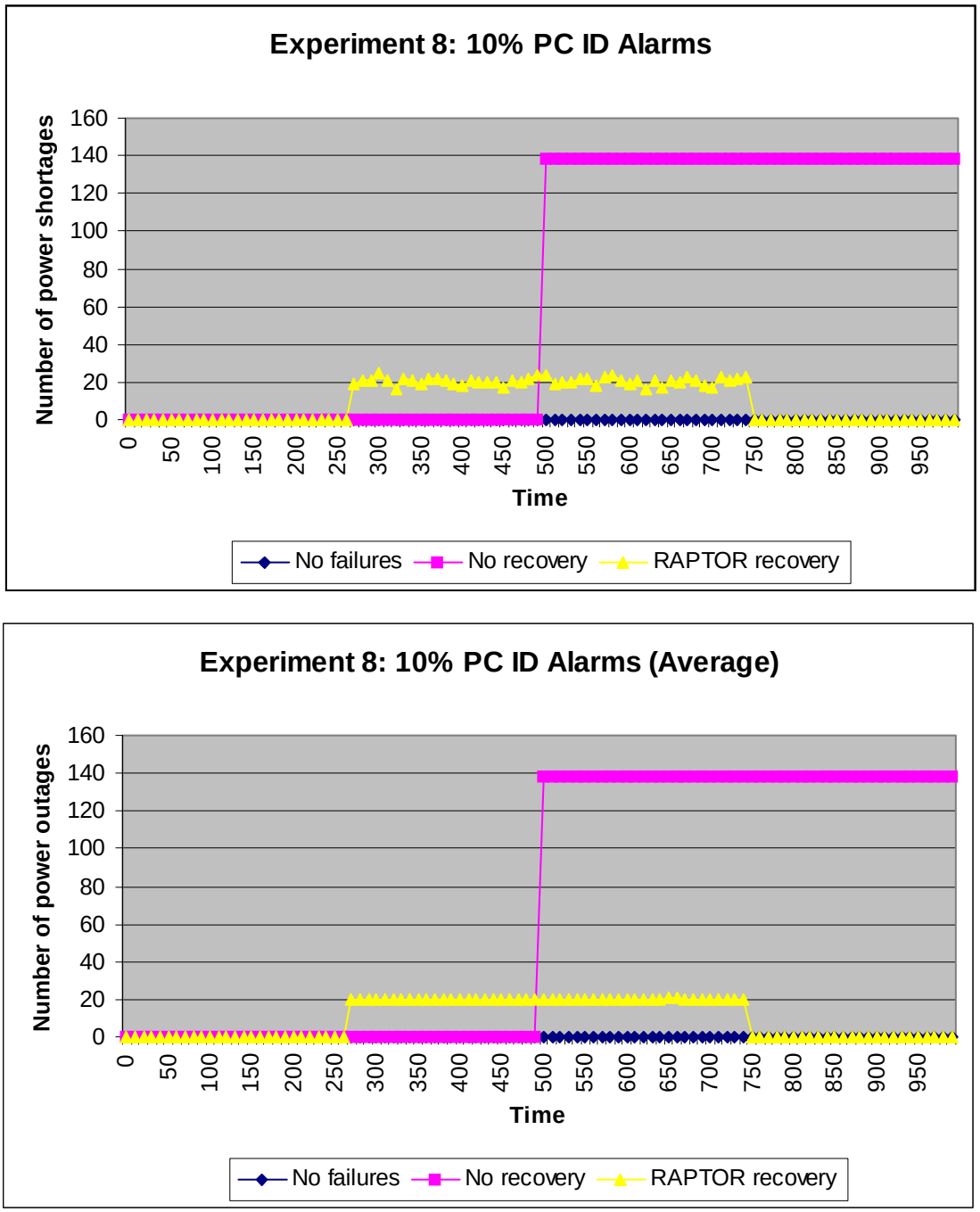

Figure 32: Experiment 8 (10% Power company intrusion detection alarms)

Experiment 8

Finally, the eighth experiment presented involves the detection of and response to a coor-

dinated security attack on a total of ten percent of the power companies, indicated by intrusion detection alarms at those nodes. Coordinated security attacks in the system are detected by the node represented by the National Infrastructure Protection Center (NIPC). However, in the system model only Control Areas and Control Regions are a part of that control system hierarchy, thus the power companies first report their intrusion detection alarms to their connected Control Area. Then, the Control Area reports a regional coordinated security attack to NIPC when two-thirds of its power companies have experienced intrusion detection alarms. Once NIPC detects a widespread coordinated security attack against power companies, it signals the affected Control Areas to restrict communications, thus taking a protective posture against further attacks. Figure 32 shows the data relating to this experiment, where the duration of the faults (intrusion detection alarms) was from time 250 to 750. As mentioned previously, an alarm does not inherently affect processing, so the *No recovery* data is unaffected. However, if the fault is not correctly diagnosed and addressed, a more significant, catastrophic event could take place, symbolized in this experiment at time 500 as the widespread power failure as the result of the compromise of power grid information systems.

Other Experiments

Again, these four experiments relating to the power system are a subset of the experiments that were run and could be run using the RAPTOR Simulator. Other non-local faults that could be detected and responded to in the power model include the following:

- Common-mode software failures involving combinations of power system nodes
- Database corruption involving combinations of power system nodes
- Other coordinated security attack patterns involving combinations of power system nodes

11.3 Analysis

While overall system performance and other statistics can be determined empirically, it is also possible to calculate analytically certain key variables. In particular, this section discusses analysis of detection time and recovery time.

11.3.1 Analysis of Detection Time

One key measurement in the RAPTOR System is the amount of time it takes to detect an error using the network of finite-state machines. The amount of time could be expressed in terms of the number of messages that must be communicated between the application and the finite-state machines for analysis before detection of the error is accomplished.

For example, for a low-level error it takes one message to accomplish error detection: the message from the application node's sensor to its control system node where the low-level error is handled.

For high-level errors it often takes multiple low-level events to occur before the high-

level error can be detected. For example, a coordinated security attack might be defined as five intrusion detection alarms over a certain period of time. The coordinated security attack is not considered to have occurred until the fifth alarm; this fifth alarm, which will be a low-level event at some node, is called the *triggering event* for a high-level error.

Another point to consider for high-level errors is that their detection is often *not* accomplished at the finite-state machine where the local events comprising them occurs. Again, take the example of a coordinated security attack: a regional coordinated security attack might be defined as a certain number of local intrusion detection alarms. While the local finite-state machines detect the local events (intrusion detection alarms), they communicate their occurrence to a regional finite-state machine in order for analysis and detection of the regional, high-level error. This communication contributes to the detection time for high-level errors.

Let us define the following variables:

n = *diameter of the control system network of finite-state machines*

m = *number of control system finite-state machines between where the triggering event is handled and where the high-level event is recognized*

Given these variables, calculation of the detection time in terms of message count is straightforward. It always takes one message to communicate the triggering event from the application sensor to the network of finite-state machines. Then, it takes *m* messages between control system finite-state machines to recognize the high-level event, so the total number of messages for error detection is $(m + 1)$.

The upper bound on the number of messages for error detection is $(n + 1)$. In the banking model the diameter of the network of finite-state machines was three, and in the power model the diameter was four. So in both of these applications, for the control systems that were constructed, detection of high-level errors required a relatively small number of messages.

11.3.2 Analysis of Recovery Time

Given the analysis presented for detection time, the analysis for error recovery proceeds in a similar manner. One difference between detection and recovery is that there is a single low-level event that triggers the occurrence of a high-level error, and therefore a single message propagating through the network of finite-state machines for this phase of detection. For recovery, it is quite possible that the recovery response requires actions at multiple application nodes, and therefore multiple messages propagating through the network of finite-state machines. That circumstance is considered in turn.

First, for a local, low-level error that requires only a local response, after error detection it takes one additional message to accomplish error recovery: the message from the control system node where the low-level error is handled to the application node's actuator prompting recovery. Thus, the total number of messages to detect and recover from a local, low-level error is two.

For high-level errors, if a response is required at the application node where the triggering event occurred, then a message must propagate back through the network of finite-state machines. This requires *m* messages to return to the originating finite-state machine,

with an upper bound of n messages. Adding the one message from the originating finite-state machine to the application node actuator, it requires $(m + 1)$ messages for a recovery response at the application node with the triggering low-level fault, with an upper bound of $(n + 1)$. In this case, the total number of messages to detect and recover from a high-level error is $2 \times (m + 1)$ with an upper bound of $2 \times (n + 1)$.

It is often the case, though, that recovery responses are required at multiple application nodes in the case of high-level errors. While it is impossible to analyze all possible recovery scenarios, one plausible scenario is that a recovery response is required at all application nodes controlled by particular finite-state machines. So, for the example of a regional finite-state machine detecting a coordinated security attack at local nodes, if there are c local finite-state machines and a application nodes controlled by those finite-state machines, then the number of recovery messages is $(m \times c + c \times a)$, where $m = 1$ in the case of regional to local communication. In general, recovery from errors will require more messages than detection of a high-level error from just the triggering event (though this is obviously not a fair comparison because many more messages relating to the high-level error could have been communicated prior to the triggering event).

12

Evaluation

This chapter addresses the research questions posed for evaluation (in Section 10.2). The key research questions related the four solution requirements to the major solution components. Those questions are now answered using the sets of experiments performed and the analysis provided in the previous chapter, as well as the solution descriptions from chapters seven through nine.

12.1 Scale

The issue of scale pervades each aspect of the solution, and as such was a major focus of this research. The following subsections discuss how scale was addressed in each solution component.

Specification

Do the specification notations and methodology enable description of large-scale systems, errors involving large numbers of nodes, and error recovery activities involving large numbers of nodes?

The System Specification enables description of large-scale systems through the abstraction and encapsulation facilities of the object-oriented notation. The number of nodes in a large-scale system is abstracted away by dealing with different node types and groups of related nodes in the specification. Nodes of the same node type and other related nodes can be grouped and treated similarly as sets, and the scale of the systems can be accommodated so long as the number of node types remains tractable. For example, in the banking model there are over 10,000 nodes, but only three distinct node types that must be specified (specific nodes are further differentiated by the values of their member variables). In the power model, there are approximately 4,500 nodes but only six distinct node types, thus simplifying the System Specification greatly.

The Error Detection Specification enables description of errors involving large num-

bers of nodes again by abstracting the large numbers away using sets involving nodes of similar types. When describing errors of interest, the large numbers of nodes can be expressed either as specific numbers or in terms of set percentages, whatever is required by the application expert. For example, a coordinated security attack could be defined either as intrusion detection alarms at 990 branch banks or intrusion detection alarms at 10% of the branch banks. In addition, errors of interest can be structured and addressed at different abstraction levels in finite-state machines as appropriate: locally, regionally, or globally. Thus, in the banking model, a coordinated security attack on a single commercial bank can be specified separately from a coordinated security attack on the entire banking network, using abstraction to deal with only as many nodes as necessary when defining faults in a large-scale system.

Similarly, the Error Recovery Specification enables description of error recovery activities involving large numbers of nodes by grouping recovery responses according to sets of related nodes, rather than requiring each node to be dealt with individually. Recovery can be addressed at local, regional, and global levels—just like detection—in order to cope with the scale of these systems. In the electric power example, local recovery responses can be effected to handle the loss of generating power within a single power company. If, however, at a regional level (within a particular control area or control region) there is a widespread loss of generating power, then a regional recovery response can be initiated.

Synthesis

Can the Fault Tolerance Translator generate sensor and actuator code for a large-scale system and its associated control system? Can the database be synthesized to accommodate a large-scale system?

The Fault Tolerance Translator can generate code for the Control System of a large-scale system because code is generated for abstract finite-state machines of the Control System based on sets of related nodes rather than for each individual node. In the banking model there are three finite-state machines corresponding to the branch bank, money-center bank, and Federal Reserve Bank levels. In the power model there are four finite-state machine types generated.

Similarly, sensors and actuators can be generated and implemented for large numbers of nodes because sensors and actuators are the same for all nodes of a particular node type, thus abstracting away the particulars of a large number of nodes.

The database can be synthesized to accommodate a large-scale system because the System Specification describes the system in terms of distinct node types. The database preprocessor can take this specification and then generate a database definition for those node types. During system operation, the database can be instantiated with objects for any or all of the nodes in a large-scale system; the database used for system description stores large numbers of objects efficiently [59]. Thus, the database can accommodate the configuration information of a large-scale system.

Implementation Architecture

Can the Control System monitor and effect control for a large number of nodes? Can the Coordinated Recovery Layer accomplish coordination between a large, distributed set of nodes?

The Control System can monitor and effect control for large-scale systems because the finite-state machines in Control System nodes monitor sets of related nodes and detect errors at different levels of abstraction. The Control System can also employ a distributed architecture to monitor and effect control more efficiently in large-scale systems, balancing the cost of communication between abstract finite-state machines with the cost of communication between the Control System and application nodes. In the power model, the Control System is distributed across 583 threads, with separate nodes at each level of control in the administrative hierarchy.

The node architecture consisting of sensors, actuators, and reconfigurable processes also facilitates monitoring and control of large numbers of nodes by providing a systematic structure and uniform interface that the Control System can utilize to effect control in all application nodes, as discussed in Section 9.1.

The Coordinated Recovery Layer can accommodate coordination in large-scale systems, between any set of distributed nodes, of any size. The Control System dictates the set of nodes to be involved in a coordinated recovery operation, and the size and membership of this set is arbitrary. In the banking model, for example, when a money-center bank finite-state machine detects the failure of its money-center bank application node, the Control System constructs a set of nodes consisting of the connected branch banks and the Federal Reserve Bank in order to coordinate the switch over to a backup facility; the Coordinated Recovery Layer uses this set when performing coordinated commitment. The size of the set for this recovery response is 100 nodes, though other recovery responses could involve larger sets of nodes.

Summary

In summary, the RAPTOR system addresses the requirement of scale from specification through implementation. Both the banking and electric power models were large-scale systems for which specifications were defined, code synthesized, and an implementation constructed. The experiments presented in the previous chapter demonstrated the ability of the RAPTOR implementation architecture to tolerate faults in large-scale systems. In addition, the faults of concern in some circumstances involved large numbers of nodes, and the recovery responses to various faults involved large numbers of nodes.

12.2 Heterogeneity

Heterogeneity is another key characteristic of critical information systems that greatly impacted the solution approach. The following subsections discuss how heterogeneity was addressed in each solution component.

Specification

Do the specification notations and methodology enable description of heterogeneous nodes, the faults involving heterogeneous nodes, and the different types of recovery response possible in those nodes?

The System Specification enables description of heterogeneous nodes using the abstraction facilities of the object-oriented notation: each node type warrants its own class description, then heterogeneity in each node type can be accomplished with parameterization of member variables. For example, in the banking model there are the three different node types, each with its own class definition, but the heterogeneity in hardware platform, operating system, geographic location, and power company is represented with four member variables that can be defined appropriately for each node. Any other arbitrary, relevant node characteristics contributing to the heterogeneity in the system can be expressed using other member variables.

The Error Detection Specification enables description of the variety of faults associated with heterogeneous nodes because different local and low-level faults can be associated with each node type in the abstract finite-state machine definitions. In the power model, the power company finite-state machine monitors substation and generator information systems in addition to their parent power company, and therefore the local faults defined for this finite-state machine are different from those for other finite-state machine types. Similarly, the Error Detection Specification permits arbitrary combinations of low-level faults to be composed to define non-local and high-level faults involving heterogeneous nodes.

The Error Recovery Specification enables description of the variety of recovery responses for heterogeneous nodes because different responses can be defined for both low-level and high-level faults at each node type. This is a necessity, as different node types perform different functions and therefore provide different alternate services. In the banking system, the alternate service of queueing checks at a branch bank does not apply at the Federal Reserve Bank, which does not deal with individual checks. The Error Recovery Specification associates different messages with different heterogeneous node types in order to define a variety of recovery responses.

Synthesis

Can the Fault Tolerance Translator generate sensors and actuators for heterogeneous nodes, as well as a control system to monitor and effect control for those nodes? Can the synthesized database accommodate the descriptions of heterogeneous nodes?

The synthesized database can accommodate the descriptions of heterogeneous nodes by storing records for each relevant node, including its particular heterogeneous characteristics. Database schemas were synthesized for the different node types, but that is only one dimension of heterogeneity. The other dimensions of heterogeneity—hardware platform, operating system, geographic location, etc.—are addressed as member variables in the database objects that are instantiated for each relevant node, as mentioned previously.

The Fault Tolerance Translator can generate code for different types of heterogeneous

node and the associated Control System because the abstract finite-state machines that were specified on a per-node-type basis are synthesized for each node type. The Control System finite-state machines then can access node characteristics in the configuration database to handle heterogeneity in the nodes during error detection analysis.

Similarly, sensors and actuators can be generated and implemented for heterogeneous nodes because they are generated for each node type, independent of the heterogeneity of the nodes into which the sensors and actuators are integrated.

Implementation Architecture

Does the reconfigurable process architecture provide a feasible structuring mechanism for the different types of heterogeneous node? Can the Control System monitor and effect control over heterogeneous nodes? Can the Coordinated Recovery Layer provide coordination services for those heterogeneous node types?

The reconfigurable process node architecture provides a feasible structuring mechanism for different types of heterogeneous node because the requirements placed upon reconfigurable processes merely supplement the different modes of functionality present in heterogeneous nodes, as discussed in Section 9.1.1. In fact, the reconfigurable process node architecture is an ideal structuring mechanism for heterogeneous nodes in the RAPTOR System, because alternate modes of functionality are structured systematically and can be manipulated easily in reconfigurable processes, regardless of the degree of heterogeneity.

The Control System can monitor and effect control over heterogeneous nodes because the heterogeneity is managed through a standard interface for control: sensors and actuators. In addition, the Control System can access the database of system configuration as part of analysis during high-level error detection. The database of system configuration provides an organization for the heterogeneous characteristics of application nodes that the Control System can utilize when necessary. For example, in the banking model, if a large set of branch banks experience node failures in rapid succession, the Control System can detect this and check the database for configuration information regarding the failed nodes. If all of the nodes have the same operating system or hardware platform, it could be evidence of a correlated fault and the Control System can act appropriately.

The Coordinated Recovery Layer can provide coordination services for heterogeneous nodes because it has a platform-independent service interface. The coordinated commitment service that was provided in both the banking and power models operates independent of the heterogeneity of the nodes involved in the protocol.

Summary

In summary, the RAPTOR System addresses the requirement of heterogeneity from specification through implementation. The nodes in both the banking and electric power models were heterogeneous in terms of application services, local faults, recovery capabilities, and other node characteristics. The specifications captured this heterogeneity amongst nodes, in addition to heterogeneity in fault types and their recovery responses. Again,

from these specifications, code was synthesized and an implementation constructed. The experiments presented in the previous chapter demonstrated two heterogeneous systems that were made to tolerate faults.

12.3 Complexity

The complexity associated with critical information systems also influences the components of the solution approach. The following subsections discuss how complexity was addressed in each solution component.

Specification

Do the specification notations and methodology enable description of the complex functionality and architecture of critical information systems, as well as the complex faults of concern and recovery responses?

The System Specification enables description of complex functionality and architecture in critical information systems using abstraction, encapsulation, and inheritance in the object-oriented notation. In the banking model, the use of inheritance simplifies the system description by collecting the information common to all bank types into a parent *Bank* class. In both models, the use of references enables description of complex, arbitrary relationships between nodes and node types, including sets of related nodes.

The Error Detection Specification enables description of complex faults of concern by structuring fault descriptions according to a network of communicating finite-state machines, as presented in Section 7.1.2. This allows local faults to be handled at the appropriate level, which helps to manage complexity in error definition. The network of abstract finite-state machines also facilitates description of non-local faults through composition of information using communication between finite-state machines. The Error Detection Specification enables explicit description of event information to be communicated, allowing the specifier to manage complexity in the analysis of non-local faults. Finally, the use of the general state-based specification notation, Z, lets the specifier describe the finite-state machines at whatever level of detail required, facilitating abstraction of the actual finite-state machine if necessary or desired to help cope with the problem of state explosion.

The Error Recovery Specification enables description of complex recovery responses through composition of recovery messages in the formal specification. To a certain extent, the complexity of local recovery responses in application nodes is abstracted away by the naming mechanism for alternate service modes. Complex recovery activities as a part of a coordinated recovery response are still possible though, and the network of finite-state machines helps control complexity here as well by structuring the responses at the appropriate level of abstraction.

Synthesis

Can the Fault Tolerance Translator generate sensors and actuators for a complex application, as well as its associated control system?

The Fault Tolerance Translator can generate a Control System for a complex application because the finite-state machine network structure manages the complexity associated with monitoring and control. Once the Error Detection and Error Recovery Specifications describing the complex detection and recovery activities have been constructed, the Fault Tolerance Translator can produce a Control System architecture in which the complexity has been organized into a systematic and regular structure.

The Fault Tolerance Translator can also generate sensors and actuators for a complex application because the generated code components have limited functionality and require additional instrumentation to interface with the application. There is little complexity associated with the operations performed by the sensors and actuators: the sensors generate messages to the Control System to notify of low-level event information, and the actuators receive messages from the Control System commanding recovery activity. These generated interface components hide the complex operations performed in the applications themselves.

Implementation Architecture

Can the Control System monitor and effect control in a complex system? Can the Coordinated Recovery Layer support coordination activities for complicated recovery responses?

The Control System can monitor and effect control in a complex system using the network of finite-state machines specified for error detection and recovery and generated by the Fault Tolerance Translator. As discussed previously, the complexity of control is abstracted some by the finite-state machine network structure. In addition, the complexity of control is dealt with by providing systematic interfaces to application nodes, sensors and actuators.

The reconfigurable process architecture itself also helps manage the complexity of the application and its control. Reconfigurable processes provide a good structuring mechanism for the implementation of complex application nodes, because the alternate modes of functionality can be structured systematically with standard interfaces to change service modes.

The Coordinated Recovery Layer can support coordination activities for complicated recovery responses using a coordinated commitment protocol. The recovery responses included coordinated switch over to backup processing sites in both models, and coordinated commitment was sufficient for this type of recovery.

Summary

In summary, the RAPTOR System addresses the requirement of complexity from specification through implementation. Both the banking and electric power models involved

complex systems for which System, Error Detection, and Error Recovery Specifications were defined. The Fault Tolerance Translator processed those specifications and generated code for the Control System, sensors, and actuators in those systems. Finally, the experiments presented in the previous chapter demonstrated an implementation architecture for both systems that tolerated faults. Both the faults of concern and the recovery responses in certain circumstances were complex, involving different nodes, combinations of local faults, and coordinated recovery responses.

12.4 Performance

Finally, performance-related issues in critical information systems were addressed by the solution approach. The following subsections discuss performance issues in each solution component.

Specification

Do the specification notations and methodology enable description of the performance-related aspects of the system, faults, and responses?

The System Specification does not address performance requirements as a special case in system description. The System Specification provides the capability to describe node performance requirements as member variables and associated values, just as any other node characteristic.

The Error Detection Specification includes the notion of time and thus faults of concern that have performance or timing elements can be described. For example, in both models, non-local faults such as "some number of intrusion detection alarms *n* over some period of time *t*" for coordinated security attacks or "some period of time *t* between failures" for cascading failures can be described, as discussed in Section 7.2.2.

The Error Recovery Specification does not include specific mechanisms for describing the performance elements of recovery responses, but recovery responses with different performance characteristics can be defined separately and used accordingly. Because the Error Recovery Specification utilizes a naming mechanism for recovery responses at the node level, any performance-related aspects of recovery must be understood as part of the recovery definition.

Synthesis

Can the Fault Tolerance Translator generate code that supports the performance-related aspects of a critical information system and its associated control system?

The Fault Tolerance Translator can generate code for the Control System that detects faults with performance-related characteristics, as described in the Error Detection Specification. The Control Systems for both application models detect non-local faults with timing characteristics, as presented in the last chapter.

The Fault Tolerance Translator generates sensor and actuator code for a critical infor-

mation system that are merely interfaces for error detection information and recovery commands. As such, the generated code is efficient in that the extent of sensor functionality is to generate messages based on low-level information, and the extent of actuator functionality is to process command messages.

Implementation Architecture

Can reconfigurable process architecture support and achieve application performance requirements during recovery and reconfiguration? Can the Control System monitor and effect control in an efficient manner? Can the Coordinated Recovery Layer provide coordination services within performance bounds?

The reconfigurable process architecture can support and achieve application performance requirements during recovery and reconfiguration, given an appropriate node architecture. Enhancements to the node architecture that would enable reconfiguration in bounded time were discussed in Section 9.1.3.

The Control System can monitor and effect control in an efficient manner, because the number of messages that must be exchanged to detect errors and prompt recovery are bounded, as discussed in the Section 11.3. In addition, the Control System will run typically on dedicated machines apart from the application, thus the only processing required of the Control System nodes involves message processing and the finite-state machine processing, both of which can be executed efficiently.

The Coordinated Recovery Layer can provide coordination services with specific performance characteristics. For example, the coordinated commitment protocol has specific performance characteristics in terms of the number of messages required, based on the number of nodes involved in the protocol.

Summary

In summary, the RAPTOR System addresses performance-related issues from specification through implementation. In both the banking and electric power models, specifications of errors with timing characteristics were described, code synthesized, and an implementation constructed. The RAPTOR implementation architecture and Control System detected and tolerated faults in an efficient manner using the network of abstract finite-state machines, as shown in the experiments and analysis of the previous chapter.

13

Related Work

Developments in several technical fields can be exploited to help deal with the problem of fault tolerance in distributed applications. This chapter reviews related work in the areas of system-level approaches to fault tolerance, fault tolerance in wide-area networking applications, reconfigurable distributed systems, and formal specification systems.

13.1 Fault Tolerance in Distributed Systems

Jalote presents an excellent framework for fault tolerance in distributed systems [37]. Jalote structures the various services and approaches to fault tolerance into levels of abstraction. The layers, from highest to lowest, of a fault-tolerant distributed system according to Jalote are shown in Figure 33. Each level of abstraction provides services for tolerating faults, and in most cases there are many mechanisms and approaches for implementing the given abstraction. At the lowest level of abstraction above the distributed system itself are the building blocks of fault tolerance, including fail-stop processors, stable storage, reliable message delivery, and synchronized clocks. One level above that is another important building block—reliable and atomic broadcast—different protocols provide different guarantees with respect to reliability, ordering, and causality of broadcast communication. The levels above that provide the services upon which systems can be built to tolerate certain types of fault, including abstractions for atomic actions and processes and data resilient to low-level failures. Finally, the highest level of abstraction enables tolerance of design faults in the software itself.

13.2 Fault-Tolerant Systems

Given this framework for fault tolerance in distributed systems, many system-level approaches exist that provide various subsets of abstractions and services. This subsection

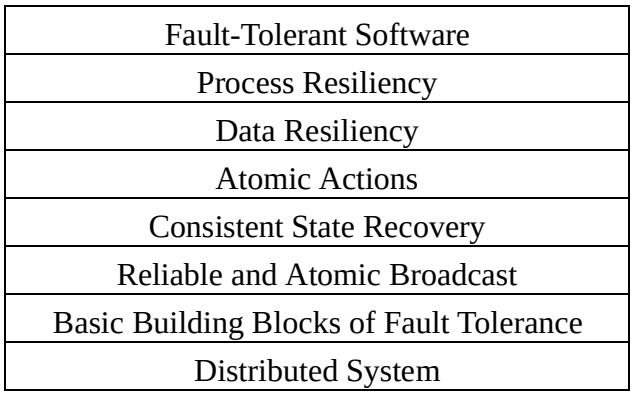

Figure 33: Levels of a fault-tolerant distributed system [37]

surveys some of the existing work on fault-tolerant system architectures.

13.2.1 Cristian/Advanced Automation System

Cristian provided a survey of the issues involved in providing fault-tolerant distributed systems [21]. He presented two requirements for a fault-tolerant system: 1) mask failures when possible, and 2) ensure clearly specified failure semantics when masking is not possible. The majority of his work, however, dealt with the masking of failures.

An instantiation of Cristian's fault tolerance concepts was used in the replacement Air Traffic Control (ATC) system, called the Advanced Automation System (AAS). The AAS utilized Cristian's fault-tolerant architecture [24]. Cristian described the primary requirement of the air traffic control system as ultra-high availability and stated that the approach taken was to design a system that can automatically mask multiple concurrent component failures.

The air traffic control system described by Cristian handled relatively low-level failures. Redundancy of components was utilized and managed in order to mask these faults. Cristian structured the fault-tolerant architecture using a "depends-on" hierarchy, and modelled the system in terms of servers, services, and a "uses" relation. Redundancy was used to mask both hardware and software failures at the highest level of abstraction, the application level. Redundancy was managed by application software server groups [24].

13.2.2 Birman/ISIS, Horus, and Ensemble

A work similar to that of Cristian is the "process-group-based computing model" presented by Birman. Birman introduced a toolkit called ISIS that contained system support for process group membership, communication, and synchronization. ISIS balanced tradeoff's in closely synchronized distributed execution (which offers easy understanding) and asynchronous execution (which achieves better performance through pipelined communi-

cation) by providing the virtual synchrony approach to group communication. ISIS facilitated group-based programming by providing a software infrastructure to support process group abstractions. Both Birman's and Cristian's work addressed a "process-group-based computing model," though Cristian's AAS also provided strong real-time guarantees made possible by an environment with strict timing properties [10].

Work on ISIS proceeded in subsequent years resulting in another group communications system, Horus. The primary benefit of Horus over ISIS was a flexible communications architecture that can be varied at runtime to match the changing requirements of the application and environment. Horus achieved this flexibility using a layered protocol architecture in which each module is responsible for a particular service [78]. Horus also worked with a system called Electra, which provided a CORBA-compliant interface to the process group abstraction in Horus [49]. Another system that built on top of Electra and Horus together, Piranha, provided high availability by supporting application monitoring and management facilities [50].

Horus was succeeded by a new tool for building adaptive distributed programs, Ensemble. Ensemble further enabled application adaptation through a stackable protocol architecture as well as system support for protocol switching. Performance improvements were also provided in Ensemble through protocol optimization and code transformations [79].

An interesting note on ISIS, Horus, and Ensemble was that all three acknowledged the security threats to the process group architecture and each incorporated a security architecture into its system [65], [66], [67].

13.2.3 Other System-level Approaches

Another example of fault tolerance that focuses on communication abstractions is the work of Schlichting, *et al*. The result of this work, a system called Coyote, supports configurable communication protocol stacks. The goals are similar to that of Horus and Ensemble, but Coyote generalizes the composition of microprotocol modules allowing non-hierarchical composition (Horus and Ensemble only support hierarchical composition). In addition, Horus and Ensemble are focusing primarily on group communication services while Coyote supports a variety of high-level network protocols [9].

Many of the systems mentioned above focus on communication infrastructure and protocols for providing fault tolerance; another approach focuses on transactions in distributed systems as the primary primitive for providing fault tolerance. One of the early systems supporting transactions was Argus, developed at MIT. Argus was a programming language and support system that defined transactions on software modules, ensuring persistence and recoverability [11].

Another transaction-based system, Arjuna, was developed at the University of Newcastle upon Tyne. Arjuna is an object-oriented programming system that provides atomic actions on objects using C++ classes [71]. The atomic actions ensure that all operations support the properties of serializability, failure atomicity, and permanence of effect.

13.2.4 Discussion

A common thread through the approach taken to fault tolerance in all of these systems is that faults are masked; each of these systems attempts to provide transparent masking of failures when a fault arises. Masking requires redundancy, and not all faults can be masked because it is not possible to build enough redundancy into a system to accommodate all faults in that way. Therefore, there will be a class of faults that these approaches to fault tolerance cannot handle because there is insufficient redundancy to tolerate them by masking.

Another interesting note is that these approaches tend to be communication-oriented. This is understandable—masking fault tolerance is dependent on redundancy, and one key to managing redundancy is maintaining consistent views of the state across all redundant entities. Supporting such a requirement within the communications framework—building guarantees into that framework—is a common approach to providing fault tolerance, but communications is not the only aspect of the system that must be addressed for a comprehensive fault-tolerance strategy.

Finally, the scale of these systems tends not to be on the order of critical information systems. Fault tolerance is applied typically to relatively small-scale systems; critical information systems are many orders of magnitude larger than the distributed systems that most of the previous work has addressed. In addition, the fault class of concern in the work described above is primarily local faults, dealing with single processor failures and limited redundancy. Local faults are not the fault model with which critical information systems and this research effort are concerned; non-local faults affecting significant portions of the network, where the redundancy to mask the fault is not available, are the faults this work addresses.

13.3 Fault Tolerance in Wide-area Network Systems

A few research efforts address fault tolerance in large-scale, wide-area network systems. In the WAFT project, Marzullo and Alvisi are concerned with the construction of fault-tolerant applications in wide-area networks. Experimental work has been done on the Nile system, a distributed computing solution for a high-energy physics project. The primary goal of the WAFT project is to adapt replication strategies for large-scale distributed applications with dynamic (unpredictable) communication properties and a requirement to withstand security attacks. Nile was implemented on top of CORBA in C++ and Java. The thrust of the work thus far is that active replication is too expensive and often unnecessary for these wide-area network applications; Marzullo and Alvisi are looking to provide support for passive replication in a toolkit [4].

The Eternal system, developed by Melliar-Smith and Moser, is middleware that operates in a CORBA environment, below a CORBA ORB but on top of their Totem group communication system. The primary goal is to provide transparent fault tolerance to users [55].

Babaoglu and Schiper are addressing problems with scaling of conventional group

technology. Their approach to providing fault tolerance in large-scale distributed systems consists of distinguishing between different roles or levels for group membership and providing different service guarantees to each level [8].

Discussion

While the approaches discussed in this section accommodate systems of a larger scale, many of the concerns raised previously still apply. These efforts still attempt to mask faults using redundancy and are primarily communications-oriented. There is still a class of faults that cannot be handled because there is insufficient redundancy.

13.4 Reconfigurable Distributed Systems

Given the body of literature on fault tolerance and the different services being provided at each abstraction layer, many types of faults can be handled. However, the most serious fault—the catastrophic, non-local fault—is not addressed by the previous related work. The previous approaches rely on having sufficient redundancy to cope with the fault and mask it; there are always going to be classes of faults for which this is not possible. For these faults, reconfiguration of the existing services on the remaining platform is required.

Considerable work has been done on reconfigurable distributed systems. Some of the work deals with reconfiguration for the purposes of evolution, as in the CONIC system. While this work is relevant, it is not directly applicable because it is concerned with reconfiguration that derives from the need to upgrade rather than cope with major faults. Less work has been done on reconfiguration for the purposes of fault tolerance. Both types of research are discussed in this section.

13.4.1 Reconfiguration Supporting System Evolution

The initial context of the work by Kramer and Magee was dynamic configuration for distributed systems, incrementally integrating and upgrading components for system evolution. CONIC, a language and distributed support system, was developed to support dynamic configuration. The language enabled specification of system configuration as well as change specifications, then the support system provided configuration tools to build the system and manage the configuration [44].

More recently, they have modelled a distributed system in terms of processes and connections, each process abstracted down to a state machine and passing messages to other processes (nodes) using the connections. One relevant finding of this work is that components must migrate to a “quiescent state” before reconfiguration to ensure consistency through the reconfiguration; basically, a quiescent state entailed not being involved in any transactions. The focus remained on the incremental changes to a distributed system configuration for evolutionary purposes [45].

The successor to CONIC, Darwin, is a configuration language that separates program structure from algorithmic behavior [54]. Darwin utilizes a component- or object-based

approach to system structure in which components encapsulate behavior behind a well-defined interface. Darwin is a declarative binding language that enables distributed programs to be constructed from hierarchically-structured specifications of component instances and their interconnections [51].

13.4.2 Reconfiguration Supporting Fault Tolerance

Purtilo developed the Polylith Software Bus, a software interconnection system that provides a module interconnection language and interfacing facilities (software toolbus). Basically, Polylith encapsulates all of the interfacing details for an application, where all software components communicate with each other through the interfaces provided by the Polylith software bus [63].

Hofmeister extended Purtilo's work by building additional primitives into Polylith for support of reconfigurable applications. Hofmeister studied the types of reconfigurations that are possible within applications and the requirements for supporting reconfiguration. Hofmeister leveraged heavily off of Polylith's interfacing and message-passing facilities in order to ensure state consistency during reconfiguration [35].

Welch and Purtilo have extended Hofmeister's work in a particular application domain, Distributed Virtual Environments. They utilized Polylith and its reconfiguration extensions in a toolkit that helps to guide the programmer in deciding on proper reconfigurations and implementations for these simulation applications [81].

13.4.3 Discussion

The research on reconfiguration for the purposes of evolution is interesting but of course does not work on the same time scale as required for fault tolerance in critical information systems. Critical information systems have performance requirements that must still be met by a fault-tolerance mechanism; reconfiguration during evolution is not concerned with performance, in general.

The existing approaches to reconfiguration for the purposes of fault tolerance, however, do not accommodate systems on the scale of critical information systems. One might argue as well that these research efforts do not handle the complexity and heterogeneity of critical information systems.

13.5 Formal Specification

A major thrust of this work is the use of a specification-based approach to fault tolerance. There are many specification notations of varying degrees of formality, intended to describe various aspects of systems and their requirements. The two primary elements described in RAPTOR specifications are an abstraction of the system itself and the finite-state machine description for fault tolerance, thus related research efforts in both system specification and finite-state machine specification are discussed.

13.5.1 System Specification

Many notations and methodologies exist for general-purpose, system specification. Most specification notations focus on a particular aspect of the system being specified; for example, functional versus non-functional requirements. Other specification systems concentrate on formalism and the ability to prove properties or animate the specification.

The systems with which we are concerned, critical information systems, are very large and complex. The goal in specifying these systems is to capture the essential elements for the purposes of tolerating faults, such as configuration, services, and other system properties.

The Unified Modeling Language (UML) is a very popular language for "specifying, visualizing, constructing, and documenting the artifacts of software systems" [64]. UML is a standards-based modeling language derived from existing languages, such as OMT, Booch, and OOSE. In addition, UML provides capabilities for business modeling and other non-software systems. UML is a visual modeling language that works particularly well with object-oriented and component-based systems [64].

UML and other modeling languages are designed to cope with complex systems, and in that respect are appropriate for system description in critical information systems. UML in particular is a very broad language, many of whose features are not required for this work. The use of the POET object-oriented database language for system description is similar to other object-oriented modeling languages though, and provides the added benefit of integration with a database.

13.5.2 Finite-State Machine Specification

The second key aspect of these systems that must be specified are the fault-tolerance requirements, including the errors to be detected and the responses to those errors. The Error Detection and Recovery Specifications centered around abstract finite-state machine definitions. Finite-state machines have been the center of many specification efforts.

Perhaps the most well-known specification methodology for finite-state machines is Statecharts [33]. Statecharts are a formalism for hierarchical finite-state machines, implemented in a software package called Statemate. Statecharts are basically finite-state machines augmented with hierarchy, parallelism, and modularity.

An extension to Statecharts, Requirements State Machine Language (RSML), is another finite-state machine-based methodology for specification [47]. RSML shares common features with Statecharts such as superstates (the grouping of states), parallel states (groupings of state machines), and state machine arrays. Enhancements in RSML included directed communication between state machines, identifier types (for transitions between states), and transition buses (for fully-interconnected states) [47].

Both Statecharts and RSML represent viable alternatives to the use of Z for state-machine specification. Both methods require explicit state-machine definition, enumerating all transitions and states, as opposed to the abstract definition made possible in a general-purpose specification language like Z. In addition, Z offers analysis benefits from formal type checkers and theorem provers. Finally, the use of Z enabled a translator to be

built that is tailored to the particulars of this problem—describing fault-tolerance requirements—and generates output that is easily integrated into the target system architecture.

14

Conclusions

This chapter concludes with a list of research contributions, explores topics for future study, and summarizes the research.

14.1 Research Contributions

There are four primary contributions of this research:

- *Focus on Application Reconfiguration for Non-Local, Catastrophic Faults.* The first contribution is a change in focus on the type of faults tolerated and the manner of handling those faults. In most fault-tolerant systems, the focus is on local faults, and the effects of those faults are masked. This work is concerned with non-local, catastrophic faults, where large and/or critical portions of the system are affected by the fault. In general, it is not possible to build sufficient redundancy into these systems to mask all faults, but especially not for masking catastrophic faults, because of the cost. The RAPTOR approach first enables description of these non-local faults in the formal specification, then supports application reconfiguration—a non-masking recovery response—using RAPTOR's application and system architecture.
- *Specification-based Approach to Fault Tolerance.* A second contribution of this work is the first use of formal specification techniques and notations to explore and describe fault tolerance in these critical information systems. The scale and complexity of most of these systems motivates the need for a formal specification approach that enables higher-level reasoning and abstraction of lower-level details in large-scale systems. RAPTOR specifications describe the system, error detection requirements, and error recovery activities using an object-oriented database language and the formal specification notation Z. The notations utilized, having precise syntax and semantics, enable various forms of analysis and further manipulation of the specifications.
- *Synthesis of Implementation Components for Fault Tolerance.* Another contribution is the construction of the Fault Tolerance Translator, which utilizes the formal specifications of fault tolerance to generate implementation components related to error detec-

tion and error recovery. High-level error detection is performed by the Control System augment to the application, and the abstract finite-state machines that perform error detection are synthesized directly from their specifications. Error recovery is effected through application actuators that accept commands from the synthesized Control System code and are themselves generated by the translator from the error recovery specification. In addition, a database of configuration information is synthesized from the system specification.

- *Implementation Architecture Supporting High-Level Error Detection and Application Reconfiguration for Error Recovery.* The final contribution of this research is a systematic structuring mechanism for high-level error detection and error recovery at the node level and system level. At the node level, there are reconfigurable processes that can be manipulated by Control System actuators to effect error recovery actions, including application reconfiguration. At the system level, there is the Control System for performing high-level error detection and coordinating error recovery, as well as the Coordinated Recovery Layer providing coordination and control services to the reconfigurable processes. These application structures facilitate an implementation of fault tolerance by enabling a regular, systematic organization rather than the typical ad hoc approach found in many systems.

14.2 Future Work

While the utility of the RAPTOR approach was successfully demonstrated on two case study applications, further study is required to fully determine the strengths and shortcomings of this approach in additional application domains. Moreover, the RAPTOR approach should be applied to a fully-functional application, rather than large-scale models and simulations. Of course, the problems that were cited in the evaluation chapter would still arise: it would require considerable resources to study actual critical information systems of the appropriate scale and complexity, but it is an important step in evaluation, if at all possible.

Another area for future study involves the linking of the fault-tolerance specification to a more formal, quantitative notion of survivability. Knight and Sullivan have proposed a formal definition of survivability based on probabilistic requirements for provision of service levels that enables testing the survivability of a system based on its survivability specification [41].

In terms of the specification notations, a more formal description of the application services provided for recovery and reconfiguration could be explored for integration into the RAPTOR specification. While it is unlikely that the entire application will be formally specified, an opportunity exists for a more formal specification of the alternate application functionality, rather than simply an enumeration of alternate services. This would then enable further analysis of the specification with respect to the levels of continued service and the interactions between alternate service modes. In addition, the use of Z for the Error Recovery Specification would enable a straightforward integration of Z specifications for application reconfiguration services.

An interesting capability of both the RAPTOR specification notations and RAPTOR Simulator is the ability to model different critical information systems in the same specification and resulting implementation. This provides the opportunity to study the interdependencies between different infrastructure application domains and model how the survivability of one system is impacted by other domains. For example, linking the banking and electric power models and simulating the effect of power system failures on the banking infrastructure would be an immensely interesting exercise and offer great insight into survivability at the level of a system of systems.

One related topic of research currently underway is the integration of reconfiguration for the purposes of fault tolerance with mechanisms for more traditional configuration management. A collaborative research effort with the University of Colorado is exploring the similarities and differences in configuration management and application reconfiguration, and a system to integrate both activities for the purposes of proactive and reactive system posturing is being built.

Finally, the security and survivability of the survivability mechanism itself is a topic of major concern. Current work at the University of Virginia addresses this topic, in part: research into running and protecting trustworthy code on possibly untrustworthy platforms is ongoing [80]. The integration of that research effort with this work is another area of future study.

14.3 Summary

This dissertation presented an approach to enable fault tolerance for the provision of survivability in critical information systems. Specific characteristics of critical information systems were described that make the requirement of achieving survivability a challenge, and the framework of fault tolerance was proposed as the mechanism to provide a survivable system. Then, the RAPTOR approach to fault tolerance was presented, a specification-based methodology for expressing high-level error detection and error recovery requirements for these systems, generating portions of the implementation specific to fault tolerance, and integrating fault-tolerance code into a reconfigurable application. Finally, the feasibility and utility of the RAPTOR approach was demonstrated for two significant case study applications.

Bibliography

[1] Adobe Systems Incorporated. “Framemaker User’s Guide,” www.adobe.com, December 2000.

[2] Agnew, B., C. Hofmeister, and J. Purtilo. “Planning for Change: A Reconfiguration Language for Distributed Systems,” Proceedings of the 2nd International Workshop on Configurable Distributed Systems, IEEE Computer Society Press, Los Alamitos, CA, March 1992, pp. 15-22.

[3] Allen, R. and D. Garlan. “Beyond Definition/Use: Architectural Interconnection,” ACM SIGPLAN Notices, Vol. 28 No. 9, August 1994, pp. 35-45.

[4] Alvisi, L. and K. Marzullo. “WAFT: Support for Fault-Tolerance in Wide-Area Object Oriented Systems,” Proceedings of the 2nd Information Survivability Workshop, IEEE Computer Society Press, Los Alamitos, CA, October 1998, pp. 5-10.

[5] Ammann, P., S. Jajodia, and P. Liu. “A Fault Tolerance Approach to Survivability,” Center for Secure Information Systems, George Mason University, April 1999.

[6] Anderson, T. and J. Knight. “A Framework for Software Fault Tolerance in Real-Time Systems,” IEEE Transactions on Software Engineering, Vol. SE-9 No. 3, May 1983, pp. 355-364.

[7] Anderson, T. and P. Lee. Fault Tolerance: Principles and Practice. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[8] Babaoglu, O. and A. Schiper. “On Group Communication in Large-Scale Distributed Systems,” ACM Operating Systems Review, Vol. 29 No. 1, January 1995, pp. 62-67.

[9] Bhatti, N., M. Hiltunen, R. Schlichting, and W. Chiu. “Coyote: A System for Constructing Fine-Grain Configurable Communication Services,” ACM Transactions on Computer Systems, Vol. 16 No. 4, November 1998, pp. 321-366.

[10] Birman, K. “The Process Group Approach to Reliable Distributed Computing,” Communications of the ACM, Vol. 36 No. 12, December 1993, pp. 37-53 and 103.

[11] Birman, K. Building Secure and Reliable Network Applications. Manning, Greenwich, CT, 1996.

[12] Blaha, M. and W. Premerlani. Object-Oriented Modeling and Design for Database Applications. Prentice Hall, Upper Saddle River, NJ, 1998.

[13] Bowen, J. “The Z notation,” http://www.afm.sbu.ac.uk/z/, April 2001.

[14] Cachin, C., J. Camenisch, M. Dacier, Y. Deswarte, J. Dobson, D. Horne, K. Kursawe, J. Laprie, J. Lebraud, D. Long, T. McCutcheon, J. Müller, F. Petzold, B. Pfitzmann, D. Powell, B. Randell, M. Schunter, V. Shoup, P. Veríssimo, G. Trouessin, R. Stroud, M. Waidner, and I. Welch. “Reference Model and Use Cases,” MAFTIA Deliverable, August 2000.

[15] Carzaniga, A., D. Rosenblum, and A. Wolf. “Content-Based Addressing and Routing: A General Model and Its Application,” Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, January 2000.

[16] Carzaniga, A., D. Rosenblum, and A. Wolf. “Design of a Scalable Event Notification Service: Interface and Architecture,” Technical Report CU-CS-863-98, Department of Computer Science, University of Colorado, August 1998.

[17] Carzaniga, A., D. Rosenblum, and A. Wolf. “Interfaces and Algorithms for a Wide-Area Event Notification Service,” Technical Report CU-CS-888-99, Department of Computer Science, University of Colorado, May 2000.

[18] “The Clinton Administration’s Policy on Critical Infrastructure Protection: Presidential Decision Directive 63,” May 1998.

[19] Cowan, C., L. Delcambre, A. Le Meur, L. Liu, D. Maier, D. McNamee, M. Miller, C. Pu, P. Wagle, and J. Walpole. “Adaptation Space: Surviving Non-Maskable Failures,” Technical Report 98-013, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, May 1998.

[20] Craigen, D., S. Gerhart, and T. Ralston. "An International Survey of Industrial Applications of Formal Methods, Volumes 1 and 2,” NIST GCR 93/626, http://hissa.ncsl.nist.gov/sw_develop/form_meth.html, March 1993.

[21] Cristian, F. “Understanding Fault-Tolerant Distributed Systems,” Communications of the ACM, Vol. 34 No. 2, February 1991, pp. 56-78.

[22] Cristian, F. and S. Mishra. “Automatic Service Availability Management in Asynchronous Distributed Systems,” Proceedings of the 2nd International Workshop on Configurable Distributed Systems, IEEE Computer Society Press, Los Alamitos, CA, March 1992, pp. 58-68.

[23] Cristian, F. “Automatic Reconfiguration in the Presence of Failures,” Software Engineering Journal, Vol. 8 No. 2, March 1993, pp. 53-60.

[24] Cristian, F., B. Dancey, and J. Dehn. “Fault-Tolerance in Air Traffic Control Systems,” ACM Transactions on Computer Systems, Vol. 14 No. 3, August 1996, pp. 265-286.

[25] Department of Energy: Energy Information Administration. “The Changing Structure of the Electric Power Industry 2000: An Update,” www.eia.doe.gov/cneaf/electricity/pubs/page.html, October 2000.

[26] Deutsch, M. and R. Willis. Software Quality Engineering: A Total Techical and Management Approach. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[27] Diller, A. Z: An Introduction to Formal Methods. John Wiley and Sons, Chichester, England, 1990.

[28] Ellison, R., D. Fisher, R. Linger, H. Lipson, T. Longstaff, and N. Mead. "Survivable Network Systems: An Emerging Discipline," Technical Report CMU/SEI-97-TR-013, Software Engineering Institute, Carnegie Mellon University, November 1997, Revised May 1999.

[29] Ellison, R., R. Linger, T. Longstaff, and N. Mead. "Survivable Network System Analysis: A Case Study," IEEE Software, Vol. 16 No. 4, July/August 1999, pp. 70-77.

[30] Federal Reserve Board. "The Twelve Federal Reserve Districts," http://www.federalreserve.gov/otherfrb.htm, March 2001.

[31] Garlan, D. and D. Perry. "Introduction to the Special Issue on Software Architecture," IEEE Transactions on Software Engineering, Vol. 21 No. 4, April 1995, pp. 269-274.

[32] Gartner, Felix C. "Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments," ACM Computing Surveys, Vol. 31 No. 1, March 1999, pp. 1-26.

[33] Harel, D. "On Visual Formalisms," Communications of the ACM, Vol. 31 No. 5, May 1988, pp. 514-530.

[34] Hofmeister, C., E. White, and J. Purtilo. "Surgeon: A Packager for Dynamically Reconfigurable Distributed Applications," Software Engineering Journal, Vol. 8 No. 2, March 1993, pp. 95-101.

[35] Hofmeister, C. "Dynamic Reconfiguration of Distributed Applications," Ph.D. Dissertation, Technical Report CS-TR-3210, Department of Computer Science, University of Maryland, January 1994.

[36] Ince, D. An Introduction to Discrete Mathematics and Formal System Specification. Clarendon Press, 1988.

[37] Jalote, P. Fault Tolerance in Distributed Systems. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[38] Knight, J., M. Elder, and X. Du. "Error Recovery in Critical Infrastructure Systems," Proceedings of Computer Security, Dependability, and Assurance '98, IEEE Computer Society Press, Los Alamitos, CA, 1999, pp. 49-71.

[39] Knight, J., M. Elder, J. Flinn, and P. Marx. "Analysis of Four Critical Infrastructure Systems," Technical Report CS-97-27, Department of Computer Science, University of Virginia, November 1997.

[40] Knight, J., R. Schutt, and K. Sullivan. "A Framework for Experimental Systems Research in Distributed Survivability Architectures," Technical Report CS-98-38, Department of Computer Science, University of Virginia, December 2000.

[41] Knight, J. and K. Sullivan. "On the Definition of Survivability," Technical Report CS-00-33, Department of Computer Science, University of Virginia, December 2000.

[42] Knight, J., K. Sullivan, X. Du, C. Wang, M. Elder, R. Lubinsky, and J. McHugh. "Enhancing Survivability of Critical Information Systems," December 1997.

[43] Knight, J. and J. Urquhart. "On the Implementation and Use of Ada on Fault-Tolerant Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-13 No. 5, May 1987, pp. 553-563.

[44] Kramer, J. and J. Magee. “Dynamic Configuration for Distributed Systems,” IEEE Transactions on Software Engineering, Vol. SE-11 No. 4, April 1985, pp. 424-436.

[45] Kramer, J. and J. Magee. “The Evolving Philosophers Problem: Dynamic Change Management,” IEEE Transactions on Software Engineering, Vol. 16 No. 11, November 1990, pp. 1293-1306.

[46] Laprie, Jean-Claude. “Dependable Computing and Fault Tolerance: Concepts and Terminology,” Digest of Papers FTCS-15: 15th International Symposium on Fault-Tolerant Computing, pp. 2-11, 1985.

[47] Leveson, N. “Requirements Specification for Process-Control System,” IEEE Transactions on Software Engineering, Vol. 20 No. 9, September 1994, pp. 684-707.

[48] Luckham, D., J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. “Specification and Analysis of System Architecture Using Rapide,” IEEE Transactions on Software Engineering, Vol. 21 No. 4, April 1995, pp. 336-355.

[49] Maffeis, S. “Electra - Making Distributed Programs Object-Oriented,” Technical Report 93-17, Department of Computer Science, University of Zurich, April 1993.

[50] Maffeis, S. “Piranha: A CORBA Tool For High Availability,” IEEE Computer, Vol. 30 No. 4, April 1997, pp. 59-66.

[51] Magee, J., N. Dulay, and J. Kramer. “Structuring Parallel and Distributed Programs,” Software Engineering Journal, Vol. 8 No. 2, March 1993, pp. 73-82.

[52] Magee, J., N. Dulay, and J. Kramer. “A Constructive Development Environment for Parallel and Distributed Programs,” Distributed Systems Engineering Journal, Vol. 1 No. 5, September 1994, pp. 304-312.

[53] Magee, J., N. Dulay, S. Eisenbach, and J. Kramer. “Specifying Distributed Software Architectures,” Lecture Notes in Computer Science, Vol. 989, September 1995, pp. 137-153.

[54] Magee, J. and J. Kramer. “Darwin: An Architectural Description Language,” http://www-dse.doc.ic.ac.uk/research/darwin/darwin.html, 1998.

[55] Melliar-Smith, P. and L. Moser. “Surviving Network Partitioning,” IEEE Computer, Vol. 31 No. 3, March 1998, pp. 62-68.

[56] Moriconi, M. and X. Qian. “Correctness and Composition of Software Architectures,” ACM SIGSOFT Software Engineering Notes, Vol. 19 No. 5, December 1994, pp. 164-174.

[57] North American Electric Reliability Council. “NERC Operating Manual,” www.nerc.com/standards, January 2001.

[58] Office of the Undersecretary of Defense for Acquisition and Technology. “Report of the Defense Science Board Task Force on Information Warfare - Defense (IW-D),” November 1996.

[59] POET Software Corporation. “POET C++ Programmer’s Guide,” May 2000.

[60] Potter, B. An Introduction to Formal Specification and Z. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[61] President’s Commission on Critical Infrastructure Protection. “Critical Foundations: Protecting America’s Infrastructures The Report of the President’s Commission on Critical Infrastructure Protection,” United States Government Printing

Office (GPO), No. 040-000-00699-1, October 1997.

[62] President's Commission on Critical Infrastructure Protection. "Report Summary—Critical Foundations: Protecting America's Infrastructures," http://www.pccip.gov, November 1997.

[63] Purtilo, J. "The POLYLITH Software Bus," ACM Transactions on Programming Languages and Systems, Vol. 16 No. 1, January 1994, pp. 151-174.

[64] Rational Software Corporation. "Unified Modeling Language Summary," www.rational.com/uml, April 2001.

[65] Reiter, M., K. Birman, and L. Gong. "Integrating Security in a Group Oriented Distributed System," Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, May 1992, pp. 18-32.

[66] Reiter, M., K. Birman, and R. van Renesse. "A Security Architecture for Fault-Tolerant Systems," ACM Transactions on Computer Systems, Vol. 12 No. 4, November 1994, pp. 340-371.

[67] Rodeh, O., K. Birman, M. Hayden, Z. Xiao, and D. Dolev. "Ensemble Security," Technical Report TR98-1703, Department of Computer Science, Cornell University, September 1998.

[68] Saaltink, Mark. "The Z/Eves User's Guide," Technical Report TR-97-5493-06, Ontario Research Associates, September 1997.

[69] Schutt, R. "The RAPTOR Simulator," http://www.cs.virginia.edu/~survive/raptor, April 2001.

[70] Shaw, M., R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. "Abstractions for Software Architecture and Tools to Support Them," IEEE Transactions on Software Engineering, Vol. 21 No. 4, April 1995, pp. 314-335.

[71] Shrivastava, S., G. Dixon, G. Parrington. "An Overview of the Arjuna Distributed Programming System," IEEE Software, Vol. 8 No. 1, January 1991, pp. 66-73.

[72] Spivey, J. "Specifying a Real-Time Kernel," IEEE Software, Vol. 7 No. 9, September 1990, pp. 21-28.

[73] Spivey, J. "Z LaTeX source and style guide," ftp://ftp.comlab.ox.ac.uk/pub/Zforum/zguide.tex.Z, ftp://ftp.comlab.ox.ac.uk/pub/Zforum/zed.sty, April 2001.

[74] Sullivan, K., J. Knight, J. McHugh, X. Du, and S. Geist. "A Framework for Experimental Systems Research in Distributed Survivability Architectures," Technical Report CS-98-38, Department of Computer Science, University of Virginia, December 1998.

[75] Sullivan, K., J. Knight, X. Du, and S. Geist. "Information Survivability Control Systems," Proceedings of the 21st International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, May 1999, pp. 184-192.

[76] Summers, B. The Payment System: Design, Management, and Supervision. International Monetary Fund, Washington, DC, 1994.

[77] UVA Software Engineering Group. "Zeus Version 2.1 User's Manual," Department of Computer Science, University of Virginia, September 2000.

[78] van Renesse, R., K. Birman, and S. Maffeis. "Horus: A Flexible Group Communications System," Communications of the ACM, Vol. 39 No. 4, April 1996, pp. 76-

83.

[79] van Renesse, R., K. Birman, M. Hayden, A. Vaysburd, and D. Karr. “Building Adaptive Systems Using Ensemble,” Technical Report TR97-1638, Department of Computer Science, Cornell University, July 1997.

[80] Wang, C. “A Security Architecture for Survivability Mechanisms,” Ph.D. Dissertation, Department of Computer Science, University of Virginia, October 2000.

[81] Welch, D. “Building Self-Reconfiguring Distributed Systems using Compensating Reconfiguration,” Proceedings of the 4th International Conference on Configurable Distributed Systems, IEEE Computer Society Press, Los Alamitos, CA, May 1998.

[82] Wing, J. “A Specifier’s Introduction to Formal Methods,” IEEE Computer, Vol. 23 No. 8, September 1990, pp. 8-24.

Appendix A: 3-Node Example/STEP

This appendix provides the specification for a simple 3-node example, described in Chapter 6 and utilized for initial problem analysis and investigation. The preliminary specification notation utilized is called STEP (Specification of the Three-node Example Program). The STEP notation presented in this appendix is strictly a *throw-away prototype* intended to illustrate some of the issues in specifying fault tolerance. The STEP notation is *not* a part of the final solution, the RAPTOR System.

The example system is a small-scale, simplified financial payments application, effecting value transfer between customer accounts. The system consists of three nodes, two branch banks and one money-center bank, and various connections between these nodes. The branch banks provide customer access (check deposit facilities) and local information storage (customer accounts), while the money-center bank maintains branch bank asset balances and routes checks for clearance between the two branch banks.

The STEP approach consists of three components—the System Architecture Specification, Service-Platform Mapping Specification, and Error Recovery Specification—that will be presented for the 3-node example in the following sections of this appendix.

A.1 System Architecture Specification

The system architecture is pictured in Figure 34. This specification component consists of a list of nodes (including attached databases) and connections. There are three nodes and five connections (primary and backup) between the various nodes.

Nodes

N1: money-center bank (MCB); attached database DB(N1)
N2: branch bank (BB); attached database DB(N2)
N3: branch bank (BB); attached database DB(N3)

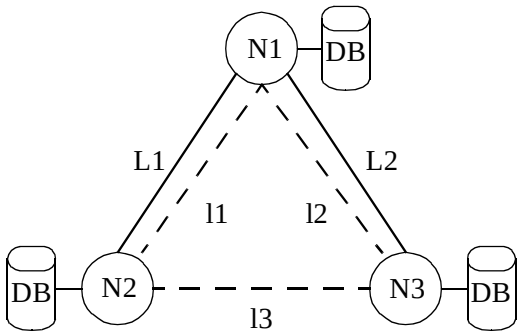

Figure 34: Example financial payments application

Connections
L1: full-bandwidth link between N1 and N2
L2: full-bandwidth link between N1 and N3
l1: low-bandwidth backup link between N1 and N2
l2: low-bandwidth backup link between N1 and N3
l3: low-bandwidth backup link between the two branch banks, N2 and N3

A.2 Service-Platform Mapping Specification

This specification component lists (names) the services that each node and connection provides, including a brief description of each service.

Money-center bank N1
MCB1: Route requests
MCB2: Maintain branch total balances (DB service)
MCB3: Buffer requests for a branch bank (Alternate)
MCB4: Send buffered requests (Alternate)
MCB5: Accept requests from customers (Alternate)

Branch banks N2, N3
BB1: Accept requests from customers, routing to other branch bank if necessary
BB2: Accept requests from other branch banks
BB3: Process requests, maintaining customer balances (DB service)
BB4: Buffer requests to pass up (Alternate)
BB5: Send buffered requests (Alternate)
BB6: Send high-priority requests, queue others (Alternate)
BB7: Pass requests directly to branch bank (Alternate)

Full-bandwidth links L1, L2
FC1: Pass full bandwidth data over full connection

Low-bandwidth backup links l1, l2
DC1: Pass limited bandwidth data over degraded connection

Low-bandwidth backup link l3
AC1: Pass limited bandwidth data over alternate connection

A.3 Error Recovery Specification

This specification component is a finite-state machine describing the system states and transitions between those states.

System States

The initial state in the finite-state machine of this example banking system consists of all the primary services operating.

S0: Initial state
N1: MCB1, MCB2
N2: BB1, BB2, BB3
N3: BB1, BB2, BB3
L1: FC1
L2: FC1

The first level of transitions from the initial state consists of a single fault occurring from the initial state. There are eight such transitions leading to the following eight states: S1, S2, S3, S4, S5, S6, S7, S8. The services after the transition include any alternate services started as a result of application reconfiguration.

S1: Process failure - MCB1 (N1)
N1: MCB2
N2: BB1, BB2, BB3, BB4
N3: BB1, BB2, BB3, BB4
L1: FC1
L2: FC1

S2: Database failure - MCB2 (N1)
N1: MCB1, MCB3
N2: BB1, BB2, BB3
N3: BB1, BB2, BB3
L1: FC1
L2: FC1

S3: Full node failure - MCB1, MCB2 (N1)
N1: -
N2: BB1, BB2, BB3, BB7
N3: BB1, BB2, BB3, BB7
L1: FC1
L2: FC1
l3: AC1

S4: Process failure - BB1 (either N2 or N3)
N1: MCB1, MCB2
N2: BB2, BB3
N3: BB1, BB2, BB3
L1: FC1
L2: FC1

S5: Process failure - BB2 (either N2 or N3)
N1: MCB1, MCB2, MCB3
N2: BB1, BB3
N3: BB1, BB2, BB3
L1: FC1
L2: FC1

S6: Database failure - BB3 (either N2 or N3)
N1: MCB1, MCB2, MCB3
N2: BB1, BB2, BB4
N3: BB1, BB2, BB3
L1: FC1
L2: FC1

S7: Full node failure - BB1, BB2, BB3 (either N2 or N3)
N1: MCB1, MCB2, MCB3
N2: -
N3: BB1, BB2, BB3
L1: FC1
L2: FC1

S8: Link failure - FC1 (either L1 or L2)
N1: MCB1, MCB2
N2: BB1, BB2, BB3, BB6
N3: BB1, BB2, BB3, BB6
L1: -
L2: FC1
l1: DC1

To handle a second sequential fault from the initial state, it is necessary to specify the list of faults that can occur from each of the above states. Then, each fault would necessitate another transition in the finite-state machine to a different state. The second level of states in the finite-state machine and the fault that causes the transition to each state are provided next, but the services after the transition are not listed here.

From State S1:
S10: Database failure - MCB2 (N1)
S11: Full node failure - MCB2 (N1)
S12: Process failure - BB1 (either N2 or N3)
S13: Process failure - BB2 (either N2 or N3)
S14: Database failure - BB3 (either N2 or N3)
S15: Process failure - BB4 (either N2 or N3)
S16: Full node failure - BB1, BB2, BB3, BB4 (either N2 or N3)
S17: Link failure - FC1 (either L1 or L2)

From State S2:

S20: Database failure - MCB2 (N1)
S21: Process failure - MCB3 (N1)
S22: Full node failure - MCB2, MCB3 (N1)
S23: Process failure - BB1 (either N2 or N3)
S24: Process failure - BB2 (either N2 or N3)
S25: Database failure - BB3 (either N2 or N3)
S26: Full node failure - BB1, BB2, BB3 (either N2 or N3)
S27: Link failure - FC1 (either L1 or L2)

From State S3:
S30: Process failure - BB1 (either N2 or N3)
S31: Process failure - BB2 (either N2 or N3)
S32: Database failure - BB3 (either N2 or N3)
S33: Process failure - BB7 (either N2 or N3)
S34: Full node failure - BB1, BB2, BB3, BB7 (either N2 or N3)
S35: Link failure - FC1 (either L1 or L2)
S36: Link failure - AC1 (l3)

From State S4:
S40: Process failure - MCB1 (N1)
S41: Database failure - MCB2 (N1)
S42: Full node failure - MCB1, MCB2 (N1)
S43: Process failure - BB2 (same N2 or N3 as previous fault)
S44: Database failure - BB3 (same N2 or N3 as previous fault)
S45: Full node failure - BB2, BB3 (same N2 or N3 as previous fault)
S46: Process failure - BB1 (different N2 or N3 from previous fault)
S47: Process failure - BB2 (different N2 or N3 from previous fault)
S48: Database failure - BB3 (different N2 or N3 from previous fault)
S49: Full node failure - BB1, BB2, BB3 (different N2 or N3 from previous fault)
S4a: Link failure - FC1 (either L1 or L2)
S4b: Link failure - AC1 (l3)

From State S5:
S50: Process failure - MCB1 (N1)
S51: Database failure - MCB2 (N1)
S52: Process failure - MCB3 (N1)
S53: Full node failure - MCB1, MCB2, MCB3 (N1)
S54: Process failure - BB1 (same N2 or N3 as previous fault)
S55: Database failure - BB3 (same N2 or N3 as previous fault)
S56: Full node failure - BB1, BB3 (same N2 or N3 as previous fault)
S57: Process failure - BB1 (different N2 or N3 from previous fault)
S58: Process failure - BB2 (different N2 or N3 from previous fault)
S59: Database failure - BB3 (different N2 or N3 from previous fault)
S5a: Full node failure - BB1, BB2, BB3 (different N2 or N3 from previous fault)
S5b: Link failure - FC1 (either L1 or L2)

From State S6:
S60: Process failure - MCB1 (N1)
S61: Database failure - MCB2 (N1)
S62: Process failure - MCB3 (N1)
S63: Full node failure - MCB1, MCB2, MCB3 (N1)

S64: Process failure - BB1 (same N2 or N3 as previous fault)
S65: Database failure - BB2 (same N2 or N3 as previous fault)
S66: Process failure - BB4 (same N2 or N3 as previous fault)
S67: Full node failure - BB1, BB2, BB4 (same N2 or N3 as previous fault)
S68: Process failure - BB1 (different N2 or N3 from previous fault)
S69: Process failure - BB2 (different N2 or N3 from previous fault)
S6a: Database failure - BB3 (different N2 or N3 from previous fault)
S6b: Full node failure - BB1, BB2, BB3 (different N2 or N3 from previous fault)
S6c: Link failure - FC1 (either L1 or L2)

From State S7:

S70: Process failure - MCB1 (N1)
S71: Database failure - MCB2 (N1)
S72: Process failure - MCB3 (N1)
S73: Full node failure - MCB1, MCB2, MCB3 (N1)
S74: Process failure - BB1 (different N2 or N3 from previous fault)
S75: Process failure - BB2 (different N2 or N3 from previous fault)
S76: Database failure - BB3 (different N2 or N3 from previous fault)
S77: Full node failure - BB1, BB2, BB3 (different N2 or N3 from previous fault)
S78: Link failure - FC1 (either L1 or L2)

From State S8:

S80: Process failure - MCB1 (N1)
S81: Database failure - MCB2 (N1)
S82: Full node failure - MCB1, MCB2 (N1)
S83: Process failure - BB1 (either N2 or N3)
S84: Process failure - BB2 (either N2 or N3)
S85: Database failure - BB3 (either N2 or N3)
S86: Process failure - BB6 (either N2 or N3)
S87: Full node failure - BB1, BB2, BB3, BB6 (either N2 or N3)
S88: Link failure - DC1 (same l1 or l2 that was started because of previous fault)
S89: Link failure - FC1 (different L1 or L2 from previous fault)

Finite-State Machine Transitions

The transitions from the initial state to the next state caused by a single fault are described in this subsection. In the state descriptions above, the remaining services after the fault and the alternate services that were started are already specified. These actions describe the handling of the fault both in response to its occurrence and after the repair of that fault (to return to the initial state):

- Transition S0 to S1:
  Response: Start BB4 (both N2 and N3)
  Repair: Start BB5 (both N2 and N3)
  Stop BB4 (both N2 and N3)
  Stop BB5 (when each queue empty)

- Transition S0 to S2:
  Response: Start MCB3
  Repair: Start MCB4
  Stop MCB3
  Stop MCB4
- Transition S0 to S3:
  Response: Start AC1
  Start BB7 (both N2 and N3)
  Repair: Stop BB7 (both N2 and N3)
  Stop AC1
- Transition S0 to S4:
  (No response)
- Transition S0 to S5:
  Response: Start MCB3
  Repair: Start MCB4
  Stop MCB3
  Stop MCB4 (when queue empty)
- Transition S0 to S6:
  Response: Start MCB3
  Start BB4 (for node N2 or N3 with fault)
  Repair: Start MCB4
  Stop MCB3
  Stop MCB4 (when queue empty)
  Stop MCB (for repaired node N2 or N3)
- Transition S0 to S7:
  Response: Start MCB3
  Repair: Start MCB4
  Stop MCB3
  Stop MCB4 (when queue empty)
- Transition S0 to S8:
  Response: Start DC1 (for failed link L1 or L2)
  Start BB6
  Repair: Start BB5
  Stop BB6
  Stop BB5 (when queue empty)
  Stop DC1 (for repaired link L1 or L2)

The transition activities for the second sequential fault are not provided here.

Appendix B: PAR System

This appendix provides the grammar for the PAR Translator and a PAR System specification for a 103-node banking example, described in Chapter 6 and utilized for initial problem analysis and investigation. The PAR (Preliminary Approach to RAPTOR) System is a preliminary approach to specifying and implementing fault tolerance. As in the previous appendix, the PAR notation presented in this appendix is strictly a *throw-away prototype* intended to illustrate some of the issues in specifying fault tolerance. The PAR System is *not* a part of the final solution, the RAPTOR System.

The PAR System consists of five specification components: the System Architecture Specification, Service-Platform Mapping Specification, System Interface Specification, Error Detection Specification, and Error Recovery Specification. The PAR System also provides a synthesizer, the PAR Translator, to process these specifications and generate implementation components. The next section provides the grammar for that translator.

B.1 Grammar for the PAR Translator

The following is the YACC grammar for the PAR Translator.

```
%%

%token SAS
%token SPMS
%token SIS
%token EDS
%token ERS
%token NODE
%token <stringtype> NODE_NAME
%token TYPE
%token <stringtype> TYPE_NAME
%token PROP
%token <stringtype> PROP_NAME
%token EVENT
%token <stringtype> EVENT_NAME
```

```
%token SERVICE
%token <stringtype> SERVICE_NAME
%token SET
%token <stringtype> SET_NAME
%token <stringtype> SET_MEMBER
%token <stringtype> VAR
%token <stringtype> COMPONENT_TYPE
%token ERROR
%token <stringtype> ERROR_NAME
%token FAILURE
%token ARROW
%token FORALL
%token EXISTS
%token AND
%token OR
%token IN
%token <inttype> START
%token <inttype> STOP

%type <inttype> critical_service

%%

par_specification
      : sas spms sis eds ers
        { printf("Parsed a PAR specification.\n"); }

sas
      : SAS declaration_list propositions

declaration_list
      : node_declarations
        { nodes.CompletedNodes(); }
      | declaration_list type_declarations
        { nodes.CompletedTypes(); }
      | declaration_list prop_declarations
        { nodes.CompletedProps(); }

node_declarations
      : node_declaration
      | node_declarations node_declaration

node_declaration
      : NODE node_list

node_list
      : NODE_NAME
        { nodes.AddNode($1); }
      | node_list ',' NODE_NAME
        { nodes.AddNode($3); }

type_declarations
      : type_declaration
      | type_declarations type_declaration

type_declaration
```

```
      : TYPE TYPE_NAME
        { nodes.AddType($2); }

prop_declarations
      : prop_declaration
      | prop_declarations prop_declaration

prop_declaration
      : PROP PROP_NAME
        { nodes.AddProp($2); }

propositions
      : proposition
      | propositions proposition

proposition
      : TYPE_NAME '(' NODE_NAME ')' ';'
        { nodes.SetNodeType($3, $1); }
      | PROP_NAME '(' NODE_NAME ')' ';'
        { nodes.SetNodeProp($3, $1); }

spms
      : SPMS service_mappings
        { nodes.CompletedNodeServiceMapping(); }

service_mappings
      : service_mapping
      | service_mappings service_mapping

service_mapping
      : single_node_service_mapping
      | multiple_node_service_mapping

single_node_service_mapping
      : NODE_NAME ARROW service_list ';'
        { nodes.MapServicesToNode($1); }

multiple_node_service_mapping
      : FORALL TYPE_NAME ARROW service_list ';'
        { nodes.MapServicesToType($2); }

service_list
      : SERVICE_NAME
        { nodes.AddService($1); }
      | service_list ',' SERVICE_NAME
        { nodes.AddService($3); }

sis
      : SIS set_declaration_list set_definitions

set_declaration_list
      : set_declarations
        { nodes.CompletedSets(); }

set_declarations
      : set_declaration
```

```
      | set_declarations set_declaration

set_declaration
      : SET SET_NAME
        { nodes.AddSet($2); }

set_definitions
      : set_definition
      | set_definitions set_definition

set_definition
      : SET_NAME '=' '{' set_list '}'
        { nodes.SetSet($1); }
      | SET_NAME '=' '{' set_iteration '}'
        { nodes.SetSet($1); }

set_list
      : SET_MEMBER
        { nodes.AddToTempSetList($1); }
      | set_list ',' SET_MEMBER
        { nodes.AddToTempSetList($3); }

set_iteration
      : VAR ':' COMPONENT_TYPE '|' PROP_NAME '(' VAR ')'
        { nodes.AddPropToTempSet($5); }
      | VAR ':' COMPONENT_TYPE '|' TYPE_NAME '(' VAR ')'
        { nodes.AddTypeToTempSet($5); }

eds
      : EDS error_declaration_list error_definitions

error_declaration_list
      : error_declarations
        { nodes.CompletedErrors(); }

error_declarations
      : error_declaration
      | error_declarations error_declaration

error_declaration
      : ERROR ERROR_NAME
        { nodes.AddError($2); }

error_definitions
      : error_definition
      | error_definitions error_definition

error_definition
      : ERROR_NAME '=' conditions ';'

conditions
      : condition
      | '(' condition ')'
      | conditions conjunction condition
      | '(' conditions conjunction condition ')'
```

```
condition
      : FAILURE '(' NODE_NAME '.' SERVICE_NAME ')'
      | EVENT_NAME '(' NODE_NAME ')'
      | '(' FORALL VAR IN SET_NAME '|' EVENT_NAME '(' VAR ')' ')'
      | '(' EXISTS VAR IN SET_NAME '|' EVENT_NAME '(' VAR ')' ')'

conjunction
      : AND
      | OR

ers
      : ERS error_activities

error_activities
      : per_error_activities
      | error_activities per_error_activities

per_error_activities
      : ERROR_NAME ':' per_node_activities
        { nodes.CompletedResponse($1); }

per_node_activities
      : per_node_responses
      | per_node_activities per_node_responses

per_node_responses
      : NODE_NAME ARROW response_list ';'
        { nodes.CompletedNodeResponse($1); }
      | SET_NAME ARROW response_list ';'
        { nodes.CompletedSetResponse($1); }

response_list
      : SERVICE_NAME '.' critical_service
        { nodes.AddResponse($1, $3); }
      | response_list ',' SERVICE_NAME '.' critical_service
        { nodes.AddResponse($3, $5); }

critical_service
      : START
        { $$ = CS_START; }
      | STOP
        { $$ = CS_STOP; }

%%
```

B.2 Example Specification: 103-Node Banking System

The following is the PAR specification of fault tolerance for a 103-node banking system.

```
SYSTEM_ARCHITECTURE_SPECIFICATION


NODE frb1, frb2, frb3
NODE mcb100, mcb200, mcb300, mcb400, mcb500, mcb600, mcb700, mcb800,
```

```
     mcb900, mcb1000
NODE bb101, bb102, bb103, bb104, bb105, bb106, bb107, bb108, bb109
NODE bb201, bb202, bb203, bb204, bb205, bb206, bb207, bb208, bb209
NODE bb301, bb302, bb303, bb304, bb305, bb306, bb307, bb308, bb309
NODE bb401, bb402, bb403, bb404, bb405, bb406, bb407, bb408, bb409
NODE bb501, bb502, bb503, bb504, bb505, bb506, bb507, bb508, bb509
NODE bb601, bb602, bb603, bb604, bb605, bb606, bb607, bb608, bb609
NODE bb701, bb702, bb703, bb704, bb705, bb706, bb707, bb708, bb709
NODE bb801, bb802, bb803, bb804, bb805, bb806, bb807, bb808, bb809
NODE bb901, bb902, bb903, bb904, bb905, bb906, bb907, bb908, bb909
NODE bb1001, bb1002, bb1003, bb1004, bb1005, bb1006, bb1007, bb1008,
     bb1009

TYPE federal_reserve
TYPE money_center
TYPE branch

PROP east_coast
PROP north_east
PROP south_east
PROP north_central
PROP south_central
PROP north_west
PROP south_west
PROP west_coast

EVENT security_attack
EVENT node_failure
EVENT power_failure

federal_reserve(frb1); federal_reserve(frb2); federal_reserve(frb3);
money_center(mcb100);
branch(bb101); branch(bb102); branch(bb103);
branch(bb104); branch(bb105); branch(bb106);
branch(bb107); branch(bb108); branch(bb109);
money_center(mcb200);
branch(bb201); branch(bb202); branch(bb203);
branch(bb204); branch(bb205); branch(bb206);
branch(bb207); branch(bb208); branch(bb209);
money_center(mcb300);
branch(bb301); branch(bb302); branch(bb303);
branch(bb304); branch(bb305); branch(bb306);
branch(bb307); branch(bb308); branch(bb309);
money_center(mcb400);
branch(bb401); branch(bb402); branch(bb403);
branch(bb404); branch(bb405); branch(bb406);
branch(bb407); branch(bb408); branch(bb409);
money_center(mcb500);
branch(bb501); branch(bb502); branch(bb503);
branch(bb504); branch(bb505); branch(bb506);
branch(bb507); branch(bb508); branch(bb509);
money_center(mcb600);
branch(bb601); branch(bb602); branch(bb603);
branch(bb604); branch(bb605); branch(bb606);
branch(bb607); branch(bb608); branch(bb609);
money_center(mcb700);
```

```
branch(bb701); branch(bb702); branch(bb703);
branch(bb704); branch(bb705); branch(bb706);
branch(bb707); branch(bb708); branch(bb709);
money_center(mcb800);
branch(bb801); branch(bb802); branch(bb803);
branch(bb804); branch(bb805); branch(bb806);
branch(bb807); branch(bb808); branch(bb809);
money_center(mcb900);
branch(bb901); branch(bb902); branch(bb903);
branch(bb904); branch(bb905); branch(bb906);
branch(bb907); branch(bb908); branch(bb909);
money_center(mcb1000);
branch(bb1001); branch(bb1002); branch(bb1003);
branch(bb1004); branch(bb1005); branch(bb1006);
branch(bb1007); branch(bb1008); branch(bb1009);

east_coast(frb1); south_east(frb2); south_central(frb3);
east_coast(mcb100);
east_coast(bb101); east_coast(bb102); north_east(bb103);
south_east(bb104); north_central(bb105); south_central(bb106);
north_west(bb107); south_west(bb108); west_coast(bb109);
east_coast(mcb200);
east_coast(bb201); east_coast(bb202); north_east(bb203);
south_east(bb204); north_central(bb205); south_central(bb206);
north_west(bb207); south_west(bb208); west_coast(bb209);
north_east(mcb300);
east_coast(bb301); north_east(bb302); north_east(bb303);
south_east(bb304); north_central(bb305); south_central(bb306);
north_west(bb307); south_west(bb308); west_coast(bb309);
south_east(mcb400);
east_coast(bb401); north_east(bb402); south_east(bb403);
south_east(bb404); north_central(bb405); south_central(bb406);
north_west(bb407); south_west(bb408); west_coast(bb409);
north_central(mcb500);
east_coast(bb501); north_east(bb502); south_east(bb503);
north_central(bb504); north_central(bb505); south_central(bb506);
north_west(bb507); south_west(bb508); west_coast(bb509);
south_central(mcb600);
east_coast(bb601); north_east(bb602); south_east(bb603);
north_central(bb604); south_central(bb605); south_central(bb606);
north_west(bb607); south_west(bb608); west_coast(bb609);
south_central(mcb700);
east_coast(bb701); north_east(bb702); south_east(bb703);
north_central(bb704); south_central(bb705); south_central(bb706);
north_west(bb707); south_west(bb708); west_coast(bb709);
north_west(mcb800);
east_coast(bb801); north_east(bb802); south_east(bb803);
north_central(bb804); south_central(bb805); north_west(bb806);
north_west(bb807); south_west(bb808); west_coast(bb809);
south_west(mcb900);
east_coast(bb901); north_east(bb902); south_east(bb903);
north_central(bb904); south_central(bb905); north_west(bb906);
south_west(bb907); south_west(bb908); west_coast(bb909);
west_coast(mcb1000);
east_coast(bb1001); north_east(bb1002); south_east(bb1003);
north_central(bb1004); south_central(bb1005); north_west(bb1006);
```

```
south_west(bb1007); west_coast(bb1008); west_coast(bb1009);


SERVICE_PLATFORM_MAPPING_SPECIFICATION


FORALL federal_reserve  ->    route_batch_requests,
                              route_batch_responses,
                              db_mc_balances,
                              frb_actuator_alert_on,
                              frb_actuator_alert_off,
                              frb_actuator_primary_frb_assignment,
                              frb_actuator_system_shutdown;

FORALL money_center     ->    route_requests,
                              route_responses,
                              db_branch_balances,
                              batch_requests,
                              send_batch_requests,
                              process_batch_requests,
                              send_batch_responses,
                              process_batch_responses,
                              mcb_actuator_alert_on,
                              mcb_actuator_alert_off,
                              mcb_actuator_new_primary_frb,
                              mcb_actuator_system_shutdown;

FORALL branch           ->    accept_requests,
                              send_requests_up,
                              db_account_balances,
                              receive_requests,
                              process_requests,
                              send_responses_up,
                              process_responses,
                              send_responses_down,
                              bb_actuator_alert_on,
                              bb_actuator_alert_off,
                              bb_actuator_system_shutdown;


SYSTEM_INTERFACE_SPECIFICATION


SET FederalReserveBanks
SET MoneyCenterBanks
SET BranchBanks
SET PrimaryFederalReserve
SET FederalReserveBackups
SET EastCoastBanks
SET NorthEastBanks
SET SouthEastBanks
SET NorthCentralBanks
SET SouthCentralBanks
SET NorthWestBanks
SET SouthWestBanks
SET WestCoastBanks
```

```
SET CitibankBanks
SET ChaseManhattanBanks

FederalReserveBanks     =     { frb1, frb2, frb3 }
MoneyCenterBanks        =     { i : NODE  |  money_center(i) }
BranchBanks             =     { i : NODE  |  branch(i) }
PrimaryFederalReserve   =     { frb1 }
FederalReserveBackups   =     { frb2, frb3 }
EastCoastBanks          =     { i : NODE  |  east_coast(i) }
NorthEastBanks          =     { i : NODE  |  north_east(i) }
SouthEastBanks          =     { i : NODE  |  south_east(i) }
NorthCentralBanks       =     { i : NODE  |  north_central(i) }
SouthCentralBanks       =     { i : NODE  |  south_central(i) }
NorthWestBanks          =     { i : NODE  |  north_west(i) }
SouthWestBanks          =     { i : NODE  |  south_west(i) }
WestCoastBanks          =     { i : NODE  |  west_coast(i) }
CitibankBanks           =     {mcb100, bb101, bb102, bb103, bb104,
                               bb105, bb106, bb107, bb108, bb109}
ChaseManhattanBanks     =     {mcb200, bb201, bb202, bb203, bb204,
                               bb205, bb206, bb207, bb208, bb209}


ERROR_DETECTION_SPECIFICATION


ERROR PrimaryFrbFailure
ERROR McbSecurityAttack
ERROR CoordinatedAttack
ERROR WidespreadPowerFailure

PrimaryFrbFailure =
      (EXISTS i IN PrimaryFederalReserve | node_failure(i)
      OR
      power_failure(i));

McbSecurityAttack =
      (EXISTS i IN MoneyCenterBanks  |  security_attack(i));

CoordinatedAttack  =
      (     (EXISTS i IN FederalReserveBanks  |  security_attack(i))
            AND
            (EXISTS i IN MoneyCenterBanks  |  security_attack(i)))
      OR
      ( FORALL i IN MoneyCenterBanks  |  security_attack(i));

WidespreadPowerFailure  =
      ( FORALL i IN EastCoastBanks  |  power_failure(i))
      OR
      ( FORALL i IN NorthEastBanks  |  power_failure(i))
      OR
      ( FORALL i IN SouthEastBanks  |  power_failure(i))
      OR
      ( FORALL i IN NorthCentralBanks  |  power_failure(i))
      OR
      ( FORALL i IN SouthCentralBanks  |  power_failure(i))
      OR
```

```
      ( FORALL i IN NorthWestBanks  |  power_failure(i))
      OR
      ( FORALL i IN SouthWestBanks  |  power_failure(i))
      OR
      ( FORALL i IN WestCoastBanks  |  power_failure(i));


ERROR_RECOVERY_SPECIFICATION


PrimaryFrbFailure(NODE): action_1
      PrimaryFrbFailure(NODE): action_1_1
            PrimaryFrbFailure(NODE): action_1_1_1
            McbSecurityAttack(NODE): action_1_1_2
            CoordinatedAttack(): action_1_1_3
            WidespreadPowerFailure(SET): action_1_1_4
      McbSecurityAttack(NODE): action_1_2
            PrimaryFrbFailure(NODE): action_1_2_1
            McbSecurityAttack(NODE): action_1_2_2
            CoordinatedAttack(): action_1_2_3
            WidespreadPowerFailure(SET): action_1_2_4
      CoordinatedAttack(): action_1_3
      WidespreadPowerFailure(SET): action_1_4
            PrimaryFrbFailure(NODE): action_1_4_1
            McbSecurityAttack(NODE): action_1_4_2
            CoordinatedAttack(): action_1_4_3
            WidespreadPowerFailure(SET): action_1_4_4

McbSecurityAttack(NODE): action_2
      PrimaryFrbFailure(NODE): action_2_1
            PrimaryFrbFailure(NODE): action_2_1_1
            McbSecurityAttack(NODE): action_2_1_2
            CoordinatedAttack(): action_2_1_3
            WidespreadPowerFailure(SET): action_2_1_4
      McbSecurityAttack(NODE): action_2_2
            PrimaryFrbFailure(NODE): action_2_2_1
            McbSecurityAttack(NODE): action_2_2_2
            CoordinatedAttack(): action_2_2_3
            WidespreadPowerFailure(SET): action_2_2_4
      CoordinatedAttack(): action_2_3
      WidespreadPowerFailure(SET): action_2_4
            PrimaryFrbFailure(NODE): action_2_4_1
            McbSecurityAttack(NODE): action_2_4_2
            CoordinatedAttack(): action_2_4_3
            WidespreadPowerFailure(SET): action_2_4_4

CoordinatedAttack(): action_3

WidespreadPowerFailure(SET): action_4
      PrimaryFrbFailure(NODE): action_4_1
            PrimaryFrbFailure(NODE): action_4_1_1
            McbSecurityAttack(NODE): action_4_1_2
            CoordinatedAttack(): action_4_1_3
            WidespreadPowerFailure(SET): action_4_1_4
      McbSecurityAttack(NODE): action_4_2
            PrimaryFrbFailure(NODE): action_4_2_1
```

```
            McbSecurityAttack(NODE): action_4_2_2
            CoordinatedAttack(): action_4_2_3
            WidespreadPowerFailure(SET): action_4_2_4
      CoordinatedAttack(): action_4_3
      WidespreadPowerFailure(SET): action_4_4
            PrimaryFrbFailure(NODE): action_4_4_1
            McbSecurityAttack(NODE): action_4_4_2
            CoordinatedAttack(): action_4_4_3
            WidespreadPowerFailure(SET): action_4_4_4


action_1(NODE frb_num):
action_1_1(NODE frb_num):
action_1_2_1(NODE frb_num):
action_1_4_1(NODE frb_num):
action_2_1(NODE frb_num):
action_2_1_1(NODE frb_num):
action_2_2_1(NODE frb_num):
action_2_4_1(NODE frb_num):
action_4_1(NODE frb_num):
action_4_1_1(NODE frb_num):
action_4_2_1(NODE frb_num):
action_4_4_1(NODE frb_num):
      frb_num -> shutdown();
      REMOVE(FederalReserveBanks, frb_num);
      REMOVE(PrimaryFederalReserve, frb_num);
      FederalReserveBanks -> reconfig_frb_down(frb_num);

action_1_1_2(NODE mcb_num):
action_1_2(NODE mcb_num):
action_1_2_2(NODE mcb_num):
action_1_4_2(NODE mcb_num):
action_2(NODE mcb_num):
action_2_1_2(NODE mcb_num):
action_2_2(NODE mcb_num):
action_2_4_2(NODE mcb_num):
action_4_1_2(NODE mcb_num):
action_4_2(NODE mcb_num):
action_4_2_2(NODE mcb_num):
action_4_4_2(NODE mcb_num):
      FederalReserveBanks -> raise_alert();
      mcb_num -> reconfig_mcb_attacked(mcb_num);

action_1_1_1(NODE frb_num):
action_1_1_3():
action_1_2_3():
action_1_3():
action_1_4_3():
action_2_1_3():
action_2_2_2(NODE mcb_num):
action_2_2_3():
action_2_3():
action_2_4_3():
action_3():
action_4_1_3():
action_4_2_3():
```

```
action_4_3():
action_4_4_3():
      BranchBanks -> shutdown();
      MoneyCenterBanks -> shutdown();
      FederalReserveBanks -> shutdown();

action_1_1_4(SET region):
action_1_2_4(SET region):
action_1_4(SET region):
action_1_4_4(SET region):
action_2_1_4(SET region):
action_2_2_4(SET region):
action_2_4(SET region):
action_2_4_4(SET region):
action_4(SET region):
action_4_1_4(SET region):
action_4_2_4(SET region):
action_4_4(SET region):
action_4_4_4(SET region):
      // if primary frb in region, promote another FRB
      // if any mcbs in region, promote BBs in other region
      switch(region)
            case east_coast:
                  frb1 -> shutdown();
                  REMOVE(FederalReserveBanks, frb1);
                  FederalReserveBanks -> reconfig_frb_down(frb1);
                  mcb100 -> shutdown();
                  REMOVE(MoneyCenterBanks, mcb100);
                  bb103 -> promote_to_mcb();
                  ADD(MoneyCenterBanks, bb103);
                  CitibankBanks -> reconfig_mcb_down(mcb100, bb103);
                  mcb200 -> shutdown();
                  REMOVE(MoneyCenterBanks, mcb200);
                  bb203 -> promote_to_mcb();
                  ADD(MoneyCenterBanks, bb203);
                  ChaseManhattanBanks -> reconfig_mcb_down(mcb200,
                                                           bb203);
            end_case;
      end_switch;
```

Appendix C: RAPTOR System

This appendix provides the grammar for the RAPTOR Fault Tolerance Translator, as well as RAPTOR specifications for the two application models described in Chapter 11: the financial payments model and the electric power model.

Each RAPTOR specification consists of two parts (described in detail in Chapter 7):

- System Specification, written in the POET database language (C++)
- Error Detection and Recovery Specifications, written in Z

The Fault Tolerance Translator processes the Error Detection and Recovery Specification and generates fault-tolerance implementation components. The next section provides the grammar for that translator.

C.1 Grammar for the RAPTOR Fault Tolerance Translator

The following is the YACC grammar for the RAPTOR Fault Tolerance Translator.

```
%%

%token Z_BEGIN
%token ZED
%token AX_DEF
%token SCHEMA
%token Z_EQUALS
%token DELTA
%token XI
%token NUM
%token NAT
%token WHERE
%token IMPLIES
%token DIV
%token POWERSET
%token FINSET
%token CARD
```

```
%token UNION
%token INTERSECTION
%token SUBSETEQ
%token SUBSET
%token MEMBEROF
%token SETMINUS
%token EMPTYSET
%token P_FUN
%token NEQ
%token SEQ
%token LANGLE
%token RANGLE
%token CAT
%token Z_END
%token BRACE_OPEN
%token BRACE_CLOSE
%token SLASHES
%token LT
%token LTE
%token GT
%token GTE
%token LOR
%token LAND
%token <value> NUMBER
%token <symbol> ID

%type <name> type
%type <name> basic_type
%type <name> expr
%type <name> id_list
%type <symbol> schema_name

%%

z_spec
      : spec_lines
        { printf("RAPTOR: GOT A COMPLETE Z SPECIFICATION.\n"); }
      ;

spec_lines
      : spec_lines spec_line
      | /* empty */
      ;

spec_line
      : Z_BEGIN '{' ZED '}' '[' given_list ']' Z_END '{' ZED '}'
        { completeGivenSets(); }
      | Z_BEGIN '{' ZED '}' ID Z_EQUALS enum_list Z_END '{' ZED '}'
        { completeSetDefn($5); }
      | Z_BEGIN '{' AX_DEF '}' declarations Z_END '{' AX_DEF '}'
        { completeAxiomDefn(); }
```

```
      | Z_BEGIN '{' AX_DEF '}' declarations WHERE conditions Z_END '{'
            AX_DEF '}'
        { completeAxiomDefn(); }
      | Z_BEGIN '{' SCHEMA '}' '{' schema_name '}' declarations Z_END '{'
            SCHEMA '}'
        { completeSchema($6); }
      | Z_BEGIN '{' SCHEMA '}' '{' schema_name '}' declarations WHERE
            conditions Z_END '{' SCHEMA '}'
        { completeSchema($6); }
      | Z_BEGIN '{' SCHEMA '}' '{' schema_name '}' declarations WHERE
            propositions Z_END '{' SCHEMA '}'
        { completeSchema($6); }
      | Z_BEGIN '{' SCHEMA '}' '{' schema_name '}' declarations WHERE
            conditions SLASHES propositions Z_END '{' SCHEMA '}'
        { completeSchema($6); }
      ;

given_list
      : given_list ',' ID
      | ID
      ;

enum_list
      : enum_list SLASHES '|' ID
        { completeIdOrder($4); }
      | enum_list '|' ID
        { completeIdOrder($3); }
      | ID
        { completeIdOrder($1); }
      ;

schema_name
      : ID
        { curr_schemaname = $1; }
      ;

declarations
      : schema_decl SLASHES var_decls
      | schema_decl
      | var_decls
      | /* empty */
      ;

schema_decl
      : schema_modifier ID
        { curr_stateschema = $2; }
      | ID
        { curr_stateschema = $1; }
      ;

schema_modifier
```

```
      : DELTA
      | XI
      ;

var_decls
      : var_decls SLASHES var_decl
      | var_decl
      ;

var_decl
      : ID ':' type
        { completeType($1, $3); }
      | ID '?' ':' type
        { completeInputType($1, $4); }
      ;

type
      : basic_type
        { $$ = $1; }
      | POWERSET basic_type
        { $$ = declareSet($2); }
      | FINSET basic_type
        { $$ = declareSet($2); }
      | ID P_FUN type
        { $$ = $1->m_name; }
      | BRACE_OPEN basic_type UNION basic_type BRACE_CLOSE
        { $$ = declareSet($2); }
      | BRACE_OPEN basic_type UNION EMPTYSET BRACE_CLOSE
        { $$ = declareSet($2); }
      ;

basic_type
      : NUM
        { $$ = NUM_STRING; }
      | NAT
        { $$ = NAT_STRING; }
      | ID
        { $$ = $1->m_name; }
      ;

conditions
      : conditions SLASHES condition
      | condition
      ;

condition
      : expr
        { outputCondition($1); }
      ;

propositions
```

```
      : propositions SLASHES proposition
      | proposition
      ;

proposition_list
      : proposition_list LAND '(' proposition ')'
      | proposition_list LAND SLASHES '(' proposition ')'
      | '(' proposition ')'
      ;

proposition
      : '(' implication ')'
      | implication
      | assignment
      ;

implication
      : implication LOR SLASHES '(' implication_clause ')'
        { addElseClause(); }
      | implication LOR SLASHES implication_clause
        { addElseClause(); }
      | '(' implication_clause ')'
      | implication_clause
      ;

implication_clause
      : expr IMPLIES '(' proposition ')'
        { assignImplication($1); }
      | expr IMPLIES SLASHES '(' proposition ')'
        { assignImplication($1); }
      | expr IMPLIES '(' proposition_list ')'
        { assignImplication($1); }
      | expr IMPLIES SLASHES '(' proposition_list ')'
        { assignImplication($1); }
      ;

assignment
      : ID '=' expr
        { assignVariable($1, $3); }
      ;

expr
      : '(' expr ')'
        { $$ = setStringParens($2); }
      | '-' expr %prec UMINUS
        { $$ = setStringMinusSign($2); }
      | expr '+' expr
        { $$ = setStringTwoOperands($1, " + ", $3); }
      | expr '-' expr
        { $$ = setStringTwoOperands($1, " - ", $3); }
      | expr '*' expr
```

```
        { $$ = setStringTwoOperands($1, " * ", $3); }
      | expr DIV expr
        { $$ = setStringTwoOperands($1, " / ", $3); }
      | expr LT expr
        { $$ = setStringTwoOperands($1, " < ", $3); }
      | expr LTE expr
        { $$ = setStringTwoOperands($1, " <= ", $3); }
      | expr GT expr
        { $$ = setStringTwoOperands($1, " > ", $3); }
      | expr GTE expr
        { $$ = setStringTwoOperands($1, " >= ", $3); }
      | expr LOR expr
        { $$ = setStringTwoOperands($1, " || ", $3); }
      | expr LOR SLASHES expr
        { $$ = setStringTwoOperands($1, " || ", $4); }
      | expr LAND expr
        { $$ = setStringTwoOperands($1, " && ", $3); }
      | expr LAND SLASHES expr
        { $$ = setStringTwoOperands($1, " && ", $4); }
      | expr UNION expr
        { $$ = setStringTwoOperands($1, UNION_STRING, $3); }
      | expr INTERSECTION expr
        { $$ = setStringTwoOperands($1, INTER_STRING, $3); }
      | expr SETMINUS expr
        { $$ = setStringTwoOperands($1, SETMI_STRING, $3); }
      | expr SUBSETEQ expr
        { $$ = setStringTwoOperands($1, SUBEQ_STRING, $3); }
      | expr MEMBEROF expr
        { $$ = setStringMemberOfSet($1, $3); }
      | expr '=' expr
        { $$ = setStringBooleanQuery($1, " == ", $3); }
      | expr NEQ expr
        { $$ = setStringBooleanQuery($1, " != ", $3); }
      | ID
        { $$ = setVariable($1); }
      | ID '?'
        { $$ = setInputVariable($1); }
      | CARD ID
        { $$ = setCardinality($2); }
      | CARD ID '?'
        { $$ = setInputCardinality($2); }
      | NUMBER
        { $$ = setNumber($1); }
      | EMPTYSET
        { $$ = EMPTYSET_STRING; }
      | BRACE_OPEN id_list BRACE_CLOSE
        { $$ = setSetIdList($2); }
      ;

id_list
      : id_list ',' ID '?'
```

```
        { $$ = setInputIdList($1, $3); }
      | id_list ',' ID
        { $$ = setIdList($1, $3); }
      | ID '?'
        { $$ = setInputVariable($1); }
      | ID
        { $$ = setVariable($1); }
      ;

%%
```

C.2 Example Specification: Financial Payments System

The next two subsections comprise the RAPTOR specification of fault tolerance for a financial payments application model.

C.2.1 System Specification

This subsection provides the class definitions of the System Specification for the banking model.

Bank Class

```
// Class definition
persistent class Bank
{
 public: // Constructors & destructor
      Bank(const int bankId, const PtString& name);
      virtual ~Bank();

 public: // Accessors
      const int BankId() const;
      const PtString& Name(void) const;
      void SetName(const PtString& name);

 public:
      bool NewObject;

 private:
      int m_bankId;
      PtString m_name;

      useindex BankIdIndex;
};


// index definition
```

```
indexdef BankIdIndex : Bank
{
      m_bankId;
};
```

FederalReserveBank Class

```
// Class definition
persistent class FederalReserveBank : public Bank
{
 public: // Constructors & destructor
      FederalReserveBank(const int bankId, const PtString& name);
      virtual ~FederalReserveBank();

 public: // Services
      void Print() const;

 public: // Accessors

 private: // Attributes
      HardwarePlatform m_HardwarePlatform;
      OperatingSystem m_OperatingSystem;
      PowerCompany* m_PowerCompany;

 public: // Pointer

 public: // Set
      lset<MoneyCenterBank*> m_SetMcbs;

 public:
      FederalReserveBankFSM m_fsm;
};


// set definition
typedef lset<FederalReserveBank*> FederalReserveBankSet;
```

MoneyCenterBank Class

```
// Class definition
persistent class MoneyCenterBank : public Bank
{
 public: // Constructors & destructor
      MoneyCenterBank(const int bankId, const PtString& name);
      virtual ~MoneyCenterBank();

 public: // Services
      void Print() const;
```

```
 public: // Accessors

 private: // Attributes
      HardwarePlatform m_HardwarePlatform;
      OperatingSystem m_OperatingSystem;
      PowerCompany* m_PowerCompany;

 public: // Pointer
      FederalReserveBank* m_Frb;

 public: // Set
      lset<BranchBank*> m_SetBbs;

 public:
      MoneyCenterBankFSM m_fsm;
};


// set definition
typedef lset<MoneyCenterBank*> MoneyCenterBankSet;
```

BranchBank Class

```
// Class definition
persistent class BranchBank : public Bank
{
 public: // Constructor & destructor
      BranchBank(const int bankId, const PtString& name);
      virtual ~BranchBank();

 public: // Services
      void Print() const;

 public: // Accessors

 private: // Attributes
      HardwarePlatform m_HardwarePlatform;
      OperatingSystem m_OperatingSystem;
      PowerCompany* m_PowerCompany;

 public: // Pointers
      MoneyCenterBank* m_Mcb;
      PowerCompany* m_PowerCompany;

 public:
      BranchBankFSM m_fsm;
};

// set definition
typedef lset<BranchBank*> BranchBankSet;
```

C.2.2 Error Detection and Recovery Specifications

This subsection contains the fault-tolerance specifications for the banking model, organized according to file type.

Declarations File

Given Sets

Z
$$[BranchBank, MoneyCenterBank, FederalReserveBank]$$

Axiomatic Descriptions

A
$$Time : \mathbb{N}$$
$$Time > 0$$

Set Definitions

Z
$$bool ::= True \mid False$$

Z
$$McbNodeStates ::= FalseMcbUp \mid Down1Mcb \mid AllMcbsDown$$

Z
$$\begin{aligned} McbBackupFrbStates ::= & \; PrimaryFrb \\ \mid & \; BackupFrb1 \\ \mid & \; BackupFrb2 \\ \mid & \; NoFrbsLeft \end{aligned}$$

Z $McbBbNodeStates ::= BbsOk \mid BbsThirdDown \mid BbsTwoThirdsDown$

Z $McbBbIdStates ::= BbsIdOk \mid BbsThirdAttacked \mid BbsTwoThirdsAttacked$

Z $FrbNodeStates ::= FalseFrbUp \mid Down1Frb \mid Down2Frbs \mid AllFrbsDown$

Z

$$
\begin{aligned}
SystemEvents ::= \; & NullEvent \\
| \; & LocalSiteFailure \\
| \; & LocalDbFailure \\
| \; & LocalPowerFailure \\
| \; & LocalIdAlarmOn \\
| \; & LocalSiteRepair \\
| \; & LocalDbRepair \\
| \; & LocalPowerRepair \\
| \; & LocalIdAlarmOff \\
| \; & BbSiteFailure \\
| \; & BbDbFailure \\
| \; & BbPowerFailure \\
| \; & BbIdAlarmOn \\
| \; & BbSiteRepair \\
| \; & BbDbRepair \\
| \; & BbPowerRepair \\
| \; & BbIdAlarmOff \\
| \; & McbSiteFailure \\
| \; & McbDbFailure \\
| \; & McbPowerFailure \\
| \; & McbIdAlarmOn \\
| \; & McbSiteRepair \\
| \; & McbDbRepair \\
| \; & McbPowerRepair \\
| \; & McbIdAlarmOff \\
| \; & McbBbsFailures \\
| \; & McbBbsRecovery \\
| \; & McbBbsIdAlarmsOn \\
| \; & McbBbsIdAlarmsOff \\
| \; & McbBbCoordinatedAttack \\
| \; & McbBbNoCoordinatedAttack \\
| \; & McbBbDbFailure \\
| \; & McbBbDbRepair \\
| \; & FrbSiteFailure \\
| \; & FrbPowerFailure \\
| \; & FrbIdAlarmOn \\
| \; & FrbSiteRepair \\
| \; & FrbPowerRepair \\
| \; & FrbIdAlarmOff \\
| \; & SystemShutdown
\end{aligned}
$$

Z

$$
\begin{aligned}
\mathit{MessagesToApplication} ::= {} & \mathit{NullBbMessage} \\
| \; & \mathit{MsgBbQueueChecks} \\
| \; & \mathit{MsgBbRaiseAlert} \\
| \; & \mathit{MsgBbSwitchToNewMcb} \\
| \; & \mathit{MsgBbBecomeBackupMcb} \\
| \; & \mathit{MsgBbShutdown} \\
| \; & \mathit{MsgBbUnQueueChecks} \\
| \; & \mathit{MsgBbLowerAlert} \\
| \; & \mathit{MsgBbStopBeingBackupMcb} \\
| \; & \mathit{NullMcbMessage} \\
| \; & \mathit{MsgMcbQueueChecks} \\
| \; & \mathit{MsgMcbRaiseAlert} \\
| \; & \mathit{MsgMcbSwitchToBackupFrb} \\
| \; & \mathit{MsgMcbSwitchBackFromBackupFrb} \\
| \; & \mathit{MsgMcbAcceptCheckRequests} \\
| \; & \mathit{MsgMcbChangeKeys} \\
| \; & \mathit{MsgMcbUnQueueChecks} \\
| \; & \mathit{MsgMcbLowerAlert} \\
| \; & \mathit{MsgMcbStopAcceptingCheckRequests} \\
| \; & \mathit{MsgMcbShutdown} \\
| \; & \mathit{NullFrbMessage} \\
| \; & \mathit{MsgFrbSwitchToBackup} \\
| \; & \mathit{MsgFrbQueueBatches} \\
| \; & \mathit{MsgFrbRaiseAlert} \\
| \; & \mathit{MsgFrbDbSnapshot} \\
| \; & \mathit{MsgFrbSwitchBackFromBackup} \\
| \; & \mathit{MsgFrbUnQueueBatches} \\
| \; & \mathit{MsgFrbLowerAlert} \\
| \; & \mathit{MsgFrbShutdown}
\end{aligned}
$$

Schema Definitions

S — *OutputEvent*

$$
\begin{array}{l}
eventname : SystemEvents \\
eventdestination : \mathbb{F}\ \mathbb{N} \\
eventtime : \mathbb{N} \\
\hline
eventname \neq NullEvent \\
eventdestination \neq \varnothing \\
eventtime \neq 0
\end{array}
$$

S — *OutputMessage*

$$
\begin{array}{l}
msgname : MessagesToApplication \\
msgdestination : \mathbb{F}\ \mathbb{N} \\
msgtime : \mathbb{N} \\
\hline
msgdestination \neq \varnothing \\
msgtime \neq 0
\end{array}
$$

State Description File

State Schemas

S *BranchBankFSM*

$$
\begin{array}{l}
MyBbs : \mathbb{F}\ \mathbb{N} \\
MyMcb : \mathbb{N} \\
BbsUp : \mathbb{F}\ \mathbb{N} \\
BbsDown : \mathbb{F}\ \mathbb{N} \\
BbsDbsDown : \mathbb{F}\ \mathbb{N} \\
BbsIdAlarms : \mathbb{F}\ \mathbb{N} \\
BbsAlarmTimes : seq\ \mathbb{N} \\
AlertAllFromMcb : bool \\
AlertAllFromBbs : bool \\
TimeAlert : bool \\
BackupMcb : bool \\
BeBackupMcb : \mathbb{N} \\
HostBb : \mathbb{N} \\
\hline
BbsUp \subseteq MyBbs \\
BbsDown \subseteq MyBbs \\
BbsUp \cup BbsDown = MyBbs \\
BbsDbsDown \subseteq MyBbs \\
BbsIdAlarms \subseteq MyBbs
\end{array}
$$

S *MoneyCenterBankFSM*

$$
\begin{array}{l}
MyMcb : \mathbb{N} \\
MyBbs : \mathbb{F}\ \mathbb{N} \\
HostBbs : \mathbb{F}\ \mathbb{N} \\
MyFrb : \mathbb{N} \\
NodesDown : McbNodeStates \\
DbDown : bool \\
Alert : bool \\
BackupFrb : McbBackupFrbStates \\
BbsUp : \mathbb{F}\ \mathbb{N} \\
BbsDown : \mathbb{F}\ \mathbb{N} \\
BbsIdAlarms : \mathbb{F}\ \mathbb{N} \\
BbsAlarmTimes : seq\ \mathbb{N} \\
BbNodeStates : McbBbNodeStates \\
BbIdStates : McbBbIdStates \\
BbCoordinatedAttack : bool \\
\hline
HostBbs \subset MyBbs \\
BbsUp \subseteq MyBbs \\
BbsDown \subseteq MyBbs \\
BbsUp \cup BbsDown = MyBbs \\
BbsIdAlarms \subseteq MyBbs
\end{array}
$$

S *FederalReserveBankFSM*

$$
\begin{array}{l}
MyFrbs : \mathbb{F}\ \mathbb{N} \\
PrimaryFrb : \mathbb{N} \\
MyMcbs : \mathbb{F}\ \mathbb{N} \\
NodesDown : FrbNodeStates \\
DbDown : bool \\
Alert : bool \\
FrbsUp : \mathbb{F}\ \mathbb{N} \\
FrbsDown : \mathbb{F}\ \mathbb{N} \\
McbsUp : \mathbb{F}\ \mathbb{N} \\
McbsDown : \mathbb{F}\ \mathbb{N} \\
McbsIdAlarms : \mathbb{F}\ \mathbb{N} \\
McbBbsDown : \mathbb{F}\ \mathbb{N} \\
McbBbsIdAlarms : \mathbb{F}\ \mathbb{N} \\
BbsCorruptDb : \mathbb{F}\ \mathbb{N} \\
McbsCorruptDb : \mathbb{F}\ \mathbb{N} \\
HalfMcbsIdAlarms : bool \\
\hline
PrimaryFrb \in MyFrbs \\
FrbsUp \subseteq MyFrbs \\
FrbsDown \subseteq MyFrbs \\
FrbsUp \cup FrbsDown = MyFrbs \\
McbsUp \subseteq MyMcbs \\
McbsDown \subseteq MyMcbs \\
McbsUp \cup McbsDown = MyMcbs \\
McbsIdAlarms \subseteq MyMcbs \\
McbBbsDown \subseteq MyMcbs \\
McbBbsIdAlarms \subseteq MyMcbs
\end{array}
$$

System State Schema

S *BankingSystemFSM*

$$
\begin{array}{l}
AllBranchBanks : \mathbb{P}\ BranchBankFSM \\
AllMoneyCenterBanks : \mathbb{P}\ MoneyCenterBankFSM \\
AllFederalReserveBanks : FederalReserveBankFSM \\
\hline
\#\ AllBranchBanks > \#\ AllMoneyCenterBanks
\end{array}
$$

Finite-State Machine File: BranchBankFSM

Initialization Schemas

S *InitBranchBankFSM*

$BranchBankFSM$

$MyBbs = \varnothing$
$MyMcb = 0$
$BbsUp = \varnothing$
$BbsDown = \varnothing$
$BbsDbsDown = \varnothing$
$BbsIdAlarms = \varnothing$
$BbsAlarmTimes = \langle\,\rangle$
$AlertAllFromMcb = False$
$AlertAllFromBbs = False$
$TimeAlert = False$
$BackupMcb = False$
$BeBackupMcb = 0$
$HostBb = 0$

S *InitBbNewBb*

$\Delta BranchBankFSM$
$bbid? : \mathbb{N}$

$MyBbs' = MyBbs \cup \{ bbid? \}$
$BbsUp' = BbsUp \cup \{ bbid? \}$

S *InitBbMyMcb*

$\Delta BranchBankFSM$
$mcbid? : \mathbb{N}$

$MyMcb' = mcbid?$

S *InitBbHostBb*

$\Delta BranchBankFSM$
$hostid? : \mathbb{N}$

$HostBb' = hostid?$

Event Schemas (Low-Level/Basic Events)

Local site failure and repair:

S *BbSchemaLocalSiteFailure*

$\Delta BranchBankFSM$
$\Delta OutputEvent$
$bbdown? : \mathbb{N}$

$BbsDown' = BbsDown \cup \{ bbdown? \}$
$BbsUp' = BbsUp \setminus \{ bbdown? \}$
$eventname' = BbSiteFailure$
$eventdestination' = \{ MyMcb \}$
$eventtime' = Time$

S *BbSchemaLocalSiteRepair*

$\Delta BranchBankFSM$
$\Delta OutputEvent$
$bbup? : \mathbb{N}$

$BbsDown' = BbsDown \setminus \{ bbup? \}$
$BbsUp' = BbsUp \cup \{ bbup? \}$
$eventname' = BbSiteRepair$
$eventdestination' = \{ MyMcb \}$
$eventtime' = Time$

Local database failure and repair:

S — *BbSchemaLocalDbFailure*

$\Delta BranchBankFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$
$bbdbdown? : \mathbb{N}$

$BbsDbsDown' = BbsDbsDown \cup \{ bbdbdown? \}$
$msgname' = MsgBbQueueChecks$
$msgdestination' = \{ bbdbdown? \}$
$msgtime' = Time$
$eventname' = BbDbFailure$
$eventdestination' = \{ MyMcb \}$
$eventtime' = Time$

S — *BbSchemaLocalDbRepair*

$\Delta BranchBankFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$
$bbdbup? : \mathbb{N}$

$BbsDbsDown' = BbsDbsDown \setminus \{ bbdbup? \}$
$msgname' = MsgBbUnQueueChecks$
$msgdestination' = \{ bbdbup? \}$
$msgtime' = Time$
$eventname' = BbDbRepair$
$eventdestination' = \{ MyMcb \}$
$eventtime' = Time$

Local power failure and repair:

S — *BbSchemaLocalPowerFailure*

$\Delta BranchBankFSM$
$\Delta OutputEvent$
$bbdown? : \mathbb{N}$

$BbsDown' = BbsDown \cup \{ bbdown? \}$
$BbsUp' = BbsUp \setminus \{ bbdown? \}$
$eventname' = BbPowerFailure$
$eventdestination' = \{ MyMcb \}$
$eventtime' = Time$

S — *BbSchemaLocalPowerRepair*

$\Delta BranchBankFSM$
$\Delta OutputEvent$
$bbup? : \mathbb{N}$

$BbsDown' = BbsDown \setminus \{ bbup? \}$
$BbsUp' = BbsUp \cup \{ bbup? \}$
$eventname' = BbPowerRepair$
$eventdestination' = \{ MyMcb \}$
$eventtime' = Time$

Local intrusion detection alarm going on and off:

S — *BbSchemaLocalIdAlarmOn*

$\Delta BranchBankFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$
$bbidalarmon? : \mathbb{N}$

$BbsIdAlarms' = BbsIdAlarms \cup \{ bbidalarmon? \}$
$BbsAlarmTimes' = BbsAlarmTimes \frown \langle Time \rangle$
$msgname' = MsgBbRaiseAlert$
$msgdestination' = \{ bbidalarmon? \}$
$msgtime' = Time$
$eventname' = BbIdAlarmOn$
$eventdestination' = \{ MyMcb \}$
$eventtime' = Time$

S — *BbSchemaLocalIdAlarmOff*

$\Delta BranchBankFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$
$bbidalarmoff? : \mathbb{N}$

$BbsIdAlarms' = BbsIdAlarms \setminus \{ bbidalarmoff? \}$
$msgname' = MsgBbLowerAlert$
$msgdestination' = \{ bbidalarmoff? \}$
$msgtime' = Time$
$eventname' = BbIdAlarmOff$
$eventdestination' = \{ MyMcb \}$
$eventtime' = Time$

Money Center Bank site failure and repair:

S *BbSchemaMcbSiteFailure*

$$
\begin{array}{l}
\Delta BranchBankFSM \\
\Delta OutputMessage \\
\hline
BackupMcb' = True \\
BeBackupMcb' = MyMcb + 1 \\
((HostBb = (MyMcb + 1)) \Rightarrow (msgname' = MsgBbBecomeBackupMcb)) \\
((HostBb \neq (MyMcb + 1)) \Rightarrow (msgname' = MsgBbSwitchToNewMcb)) \\
msgdestination' = \{ HostBb \} \\
msgtime' = Time \\
msgname' = MsgBbSwitchToNewMcb \\
msgdestination' = BbsUp \setminus \{ HostBb \} \\
msgtime' = Time
\end{array}
$$

S *BbSchemaMcbSiteRepair*

$$
\begin{array}{l}
\Delta BranchBankFSM \\
\Delta OutputMessage \\
\hline
BackupMcb' = False \\
BeBackupMcb' = 0 \\
((HostBb = (MyMcb + 1)) \Rightarrow \\
\quad ((msgname' = MsgBbStopBeingBackupMcb) \wedge \\
\quad (msgdestination' = \{ HostBb \}) \wedge (msgtime' = Time))) \\
msgname' = MsgBbSwitchToNewMcb \\
msgdestination' = BbsUp \cup \{ HostBb \} \\
msgtime' = Time
\end{array}
$$

Money Center Bank power failure and repair:

S — *BbSchemaMcbPowerFailure*

$$
\begin{array}{l}
\Delta BranchBankFSM \\
\Delta OutputMessage \\
\hline
BackupMcb' = True \\
BeBackupMcb' = MyMcb + 1 \\
((HostBb = (MyMcb + 1)) \Rightarrow (msgname' = MsgBbBecomeBackupMcb)) \\
((HostBb \neq (MyMcb + 1)) \Rightarrow (msgname' = MsgBbSwitchToNewMcb)) \\
msgdestination' = \{ HostBb \} \\
msgtime' = Time \\
msgname' = MsgBbSwitchToNewMcb \\
msgdestination' = BbsUp \setminus \{ HostBb \} \\
msgtime' = Time
\end{array}
$$

S — *BbSchemaMcbPowerRepair*

$$
\begin{array}{l}
\Delta BranchBankFSM \\
\Delta OutputMessage \\
\hline
BackupMcb' = False \\
BeBackupMcb' = 0 \\
(HostBb = (MyMcb + 1)) \Rightarrow \\
((msgname' = MsgBbStopBeingBackupMcb) \wedge \\
(msgdestination' = \{ HostBb \}) \wedge (msgtime' = Time)) \\
msgname' = MsgBbSwitchToNewMcb \\
msgdestination' = BbsUp \cup \{ HostBb \} \\
msgtime' = Time
\end{array}
$$

Money Center Bank intrustion detection alarm going on and off:

S — *BbSchemaMcbIdAlarmOn*

$$
\begin{array}{l}
\Delta BranchBankFSM \\
\Delta OutputMessage \\
\hline
AlertAllFromMcb' = True \\
msgname' = MsgBbRaiseAlert \\
msgdestination' = MyBbs \\
msgtime' = Time
\end{array}
$$

S *BbSchemaMcbIdAlarmOff*

$\Delta BranchBankFSM$
$\Delta OutputMessage$

$AlertAllFromMcb' = False$
$msgname' = MsgBbLowerAlert$
$msgdestination' = MyBbs$
$msgtime' = Time$

System shutting down:

S *BbSchemaSystemShutdown*

$\Xi BranchBankFSM$
$\Delta OutputMessage$

$msgname' = MsgBbShutdown$
$msgdestination' = MyBbs$
$msgtime' = Time$

Event Schemas (High-level Events)

Half of these Branch Bank intrusion detection alarms going on and off:

S *BbConditionSchemaHalfIdAlarmsOn*

$\Delta BranchBankFSM$
$\Delta OutputMessage$

$(AlertAllFromBbs = False) \land (\# BbsIdAlarms \geq (\# MyBbs \backslash div\ 2))$
$AlertAllFromBbs' = True$
$msgname' = MsgBbRaiseAlert$
$msgdestination' = MyBbs$
$msgtime' = Time$

$$
\begin{array}{|l}
\hline
S \quad BbConditionSchemaHalfIdAlarmsOff \\
\Delta BranchBankFSM \\
\Delta OutputMessage \\
\hline
(AlertAllFromBbs = True) \land (\#BbsIdAlarms < (\#MyBbs \setminus div\ 2)) \\
AlertAllFromBbs' = False \\
msgname' = MsgBbLowerAlert \\
msgdestination' = MyBbs \\
msgtime' = Time \\
\hline
\end{array}
$$

5 Branch bank intrusion detection alarms have gone off in the past 30 seconds, then not:

$$
\begin{array}{|l}
\hline
S \quad BbConditionSchema5AlarmsIn30Seconds \\
\Delta BranchBankFSM \\
\Delta OutputMessage \\
\hline
(TimeAlert = False) \land (\#BbsAlarmTimes \geq 5) \land \\
(BbsAlarmTimes\ ((\#BbsAlarmTimes) - 4) \geq (Time - 30)) \\
TimeAlert' = True \\
msgname' = MsgBbRaiseAlert \\
msgdestination' = MyBbs \\
msgtime' = Time \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
S \quad BbConditionSchemaNo5AlarmsIn30Seconds \\
\Delta BranchBankFSM \\
\Delta OutputMessage \\
\hline
(TimeAlert = True) \land \\
(BbsAlarmTimes\ ((\#BbsAlarmTimes) - 4) < (Time - 30)) \\
TimeAlert' = False \\
msgname' = MsgBbLowerAlert \\
msgdestination' = MyBbs \\
msgtime' = Time \\
\hline
\end{array}
$$

Finite-State Machine File: MoneyCenterBankFSM

Initialization Schemas

S *InitMoneyCenterBankFSM*

$MoneyCenterBankFSM$

$MyMcb = 0$
$MyBbs = \varnothing$
$HostBbs = \varnothing$
$MyFrb = 0$
$NodesDown = FalseMcbUp$
$DbDown = False$
$Alert = False$
$BackupFrb = PrimaryFrb$
$BbsUp = \varnothing$
$BbsDown = \varnothing$
$BbsIdAlarms = \varnothing$
$BbsAlarmTimes = \langle\ \rangle$
$BbNodeStates = BbsOk$
$BbIdStates = BbsIdOk$
$BbCoordinatedAttack = False$

S *InitMcbMyMcb*

$\Delta MoneyCenterBankFSM$
$mcbid? : \mathbb{N}$

$MyMcb' = mcbid?$

S *InitMcbNewBb*

$\Delta MoneyCenterBankFSM$
$bbid? : \mathbb{N}$

$MyBbs' = MyBbs \cup \{\ bbid?\ \}$
$BbsUp' = BbsUp \cup \{\ bbid?\ \}$

S — *InitMcbNewHostBb*

$$\Delta MoneyCenterBankFSM$$
$$bbid? : \mathbb{N}$$

$$HostBbs' = HostBbs' \cup \{ bbid? \}$$

S — *InitMcbMyFrb*

$$\Delta MoneyCenterBankFSM$$
$$frbid? : \mathbb{N}$$

$$MyFrb' = frbid?$$

Event Schemas (Low-Level/Basic Events)

Local site failure and repair:

S — *McbSchemaLocalSiteFailure*

$$\Delta MoneyCenterBankFSM$$
$$\Delta OutputEvent$$

$$((NodesDown = FalseMcbUp) \Rightarrow (NodesDown' = Down1Mcb)) \lor$$
$$((NodesDown = Down1Mcb) \Rightarrow (NodesDown' = AllMcbsDown)))$$
$$eventname' = McbSiteFailure$$
$$eventdestination' = HostBbs \cup \{ MyFrb \}$$
$$eventtime' = Time$$

S — *McbSchemaLocalSiteRepair*

$$\Delta MoneyCenterBankFSM$$
$$\Delta OutputEvent$$

$$((NodesDown = AllMcbsDown) \Rightarrow (NodesDown' = Down1Mcb)) \lor$$
$$((NodesDown = Down1Mcb) \Rightarrow (NodesDown' = FalseMcbUp)))$$
$$eventname' = McbSiteRepair$$
$$eventdestination' = HostBbs \cup \{ MyFrb \}$$
$$eventtime' = Time$$

Local database failure and repair:

S — *McbSchemaLocalDbFailure*

$\Delta MoneyCenterBankFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$

$DbDown' = True$
$msgname' = MsgMcbQueueChecks$
$msgdestination' = \{ MyMcb \}$
$msgtime' = Time$
$eventname' = McbDbFailure$
$eventdestination' = \{ MyFrb \}$
$eventtime' = Time$

S — *McbSchemaLocalDbRepair*

$\Delta MoneyCenterBankFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$

$DbDown' = False$
$msgname' = MsgMcbUnQueueChecks$
$msgdestination' = \{ MyMcb \}$
$msgtime' = Time$
$eventname' = McbDbRepair$
$eventdestination' = \{ MyFrb \}$
$eventtime' = Time$

Local power failure and repair:

S — *McbSchemaLocalPowerFailure*

$\Delta MoneyCenterBankFSM$
$\Delta OutputEvent$

$((NodesDown = FalseMcbUp) \Rightarrow (NodesDown' = Down1Mcb)) \land$
$((NodesDown = Down1Mcb) \Rightarrow (NodesDown' = AllMcbsDown))$
$eventname' = McbPowerFailure$
$eventdestination' = HostBbs \cup \{ MyFrb \}$
$eventtime' = Time$

S — *McbSchemaLocalPowerRepair*

$\Delta MoneyCenterBankFSM$
$\Delta OutputEvent$

$(((NodesDown = AllMcbsDown) \Rightarrow (NodesDown' = Down1Mcb)) \lor$
$((NodesDown = Down1Mcb) \Rightarrow (NodesDown' = FalseMcbUp)))$
$eventname' = McbPowerRepair$
$eventdestination' = HostBbs \cup \{ MyFrb \}$
$eventtime' = Time$

Local intrusion detection alarm going on and off:

S — *McbSchemaLocalIdAlarmOn*

$\Delta MoneyCenterBankFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$

$Alert' = True$
$eventname' = McbIdAlarmOn$
$eventdestination' = HostBbs \cup \{ MyFrb \}$
$eventtime' = Time$
$msgname' = MsgMcbRaiseAlert$
$msgdestination' = \{ MyMcb \}$
$msgtime' = Time$

S — *McbSchemaLocalIdAlarmOff*

$\Delta MoneyCenterBankFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$

$Alert' = False$
$eventname' = McbIdAlarmOff$
$eventdestination' = HostBbs \cup \{ MyFrb \}$
$eventtime' = Time$
$msgname' = MsgMcbLowerAlert$
$msgdestination' = \{ MyMcb \}$
$msgtime' = Time$

Federal Reserve Bank site failure and repair:

S *McbSchemaFrbSiteFailure*

$\Delta MoneyCenterBankFSM$
$\Delta OutputMessage$

$((BackupFrb = PrimaryFrb) \Rightarrow (BackupFrb' = BackupFrb1)) \lor$
$((BackupFrb = BackupFrb1) \Rightarrow (BackupFrb' = BackupFrb2)) \lor$
$((BackupFrb = BackupFrb2) \Rightarrow (BackupFrb' = NoFrbsLeft)))$
$msgname' = MsgMcbSwitchToBackupFrb$
$msgdestination' = \{ MyMcb \}$
$msgtime' = Time$

S *McbSchemaFrbSiteRepair*

$\Delta MoneyCenterBankFSM$
$\Delta OutputMessage$

$((BackupFrb = NoFrbsLeft) \Rightarrow (BackupFrb' = BackupFrb2)) \lor$
$((BackupFrb = BackupFrb2) \Rightarrow (BackupFrb' = BackupFrb1)) \lor$
$((BackupFrb = BackupFrb1) \Rightarrow (BackupFrb' = PrimaryFrb)))$
$msgname' = MsgMcbSwitchBackFromBackupFrb$
$msgdestination' = \{ MyMcb \}$
$msgtime' = Time$

Federal Reserve Bank power failure and repair:

S *McbSchemaFrbPowerFailure*

$\Delta MoneyCenterBankFSM$
$\Delta OutputMessage$

$((BackupFrb = PrimaryFrb) \Rightarrow (BackupFrb' = BackupFrb1)) \lor$
$((BackupFrb = BackupFrb1) \Rightarrow (BackupFrb' = BackupFrb2)) \lor$
$((BackupFrb = BackupFrb2) \Rightarrow (BackupFrb' = NoFrbsLeft)))$
$msgname' = MsgMcbSwitchToBackupFrb$
$msgdestination' = \{ MyMcb \}$
$msgtime' = Time$

S *McbSchemaFrbPowerRepair*

$$
\begin{array}{l}
\Delta MoneyCenterBankFSM \\
\Delta OutputMessage \\
\hline
(((BackupFrb = NoFrbsLeft) \Rightarrow (BackupFrb' = BackupFrb2)) \lor \\
\ ((BackupFrb = BackupFrb2) \Rightarrow (BackupFrb' = BackupFrb1)) \lor \\
\ ((BackupFrb = BackupFrb1) \Rightarrow (BackupFrb' = PrimaryFrb))) \\
msgname' = MsgMcbSwitchBackFromBackupFrb \\
msgdestination' = \{\ MyMcb\ \} \\
msgtime' = Time
\end{array}
$$

Federal Reserve Bank intrusion detection alarm going on and off:

S *McbSchemaFrbIdAlarmOn*

$$
\begin{array}{l}
\Delta MoneyCenterBankFSM \\
\Delta OutputMessage \\
\hline
Alert' = True \\
msgname' = MsgMcbRaiseAlert \\
msgdestination' = \{\ MyMcb\ \} \\
msgtime' = Time
\end{array}
$$

S *McbSchemaFrbIdAlarmOff*

$$
\begin{array}{l}
\Delta MoneyCenterBankFSM \\
\Delta OutputMessage \\
\hline
Alert' = False \\
msgname' = MsgMcbLowerAlert \\
msgdestination' = \{\ MyMcb\ \} \\
msgtime' = Time
\end{array}
$$

Branch Bank site failure and repair:

S *McbSchemaBbSiteFailure*

$$
\begin{array}{l}
\Delta MoneyCenterBankFSM \\
bbdown? : \mathbb{N} \\
\hline
BbsDown' = BbsDown \cup \{\ bbdown?\ \} \\
BbsUp' = BbsUp \setminus \{\ bbdown?\ \}
\end{array}
$$

S *McbSchemaBbSiteRepair*

$\Delta MoneyCenterBankFSM$
$bbup? : \mathbb{N}$

$BbsDown' = BbsDown \setminus \{ bbup? \}$
$BbsUp' = BbsUp \cup \{ bbup? \}$

Branch Bank power failure and repair:

S *McbSchemaBbPowerFailure*

$\Delta MoneyCenterBankFSM$
$bbdown? : \mathbb{N}$

$BbsDown' = BbsDown \cup \{ bbdown? \}$
$BbsUp' = BbsUp \setminus \{ bbdown? \}$

S *McbSchemaBbPowerRepair*

$\Delta MoneyCenterBankFSM$
$bbup? : \mathbb{N}$

$BbsDown' = BbsDown \setminus \{ bbup? \}$
$BbsUp' = BbsUp \cup \{ bbup? \}$

Branch Bank database failure and repair:

S *McbSchemaBbDbFailure*

$\Delta MoneyCenterBankFSM$
$\Delta OutputEvent$
$bbdbdown? : \mathbb{N}$

$eventname' = McbBbDbFailure$
$eventdestination' = \{ MyFrb \}$
$eventtime' = Time$

S — *McbSchemaBbDbRepair*

$\Delta MoneyCenterBankFSM$
$\Delta OutputEvent$
$bbdbup? : \mathbb{N}$

$eventname' = McbBbDbRepair$
$eventdestination' = \{ MyFrb \}$
$eventtime' = Time$

Branch Bank intrusion detection alarm going on and off:

S — *McbSchemaBbIdAlarmOn*

$\Delta MoneyCenterBankFSM$
$bbidalarmon? : \mathbb{N}$

$BbsIdAlarms' = BbsIdAlarms \cup \{ bbidalarmon? \}$
$BbsAlarmTimes' = BbsAlarmTimes \frown \langle Time \rangle$

S — *McbSchemaBbIdAlarmOff*

$\Delta MoneyCenterBankFSM$
$bbidalarmoff? : \mathbb{N}$

$BbsIdAlarms' = BbsIdAlarms \cup \{ bbidalarmoff? \}$

System shutting down:

S — *McbSchemaSystemShutdown*

$\Xi MoneyCenterBankFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$

$msgname' = MsgMcbShutdown$
$msgdestination' = \{ MyMcb \}$
$msgtime' = Time$
$eventname' = SystemShutdown$
$eventdestination' = HostBbs$
$eventtime' = Time$

Event Schemas (High-level Events)

One-third and two-thirds of these Branch Banks have failed and recovered:

S — *McbConditionSchemaBbsThirdDown*

$$
\begin{array}{l}
\Delta MoneyCenterBankFSM \\
\Delta OutputMessage \\
\Delta OutputEvent \\
\hline
((BbNodeStates = BbsOk) \land (\#BbsDown \geq (\#MyBbs \setminus div\ 3))) \lor \\
\quad ((BbNodeStates = BbsTwoThirdsDown) \land \\
\qquad (\#BbsDown < (2 * (\#MyBbs) \setminus div\ 3))) \\
((BbNodeStates = BbsOk) \Rightarrow \\
\quad ((msgname' = MsgMcbAcceptCheckRequests) \land \\
\qquad (msgdestination' = \{\ MyMcb\ \}) \land \\
\qquad (msgtime' = Time))) \\
((BbNodeStates = BbsTwoThirdsDown) \Rightarrow \\
\quad ((eventname' = McbBbsRecovery) \land \\
\qquad (eventdestination' = \{\ MyFrb\ \}) \land \\
\qquad (eventtime' = Time))) \\
BbNodeStates' = BbsThirdDown
\end{array}
$$

S — *McbConditionSchemaBbsTwoThirdsDown*

$$
\begin{array}{l}
\Delta MoneyCenterBankFSM \\
\Delta OutputEvent \\
\hline
(BbNodeStates = BbsThirdDown) \land (\#BbsDown \geq (2 * (\#MyBbs) \setminus div\ 3)) \\
BbNodeStates' = BbsTwoThirdsDown \\
eventname' = McbBbsFailures \\
eventdestination' = \{\ MyFrb\ \} \\
eventtime' = Time
\end{array}
$$

S — *McbConditionSchemaBbsDownRecovered*

$$
\begin{array}{l}
\Delta MoneyCenterBankFSM \\
\Delta OutputMessage \\
\hline
(BbNodeStates = BbsThirdDown) \land (\#BbsDown < (\#MyBbs \setminus div\ 3)) \\
BbNodeStates' = BbsOk \\
msgname' = MsgMcbStopAcceptingCheckRequests \\
msgdestination' = \{\ MyMcb\ \} \\
msgtime' = Time
\end{array}
$$

One-third and two-thirds of these Branch Banks have been attacked and recovered:

S — *McbConditionSchemaBbsThirdAttacked*

$$
\begin{array}{l}
\Delta MoneyCenterBankFSM \\
\Delta OutputMessage \\
\Delta OutputEvent \\
\hline
((BbIdStates = BbsIdOk) \land (\#\, BbsIdAlarms \geq (\#\, MyBbs \setminus div\ 3))) \lor \\
\quad ((BbIdStates = BbsTwoThirdsAttacked) \land \\
\quad\quad (\#\, BbsIdAlarms < (2 * (\#\, MyBbs \setminus div\ 3)))) \\
(BbIdStates = BbsIdOk) \Rightarrow \\
\quad ((msgname' = MsgMcbRaiseAlert) \land (msgdestination' = \{\ MyMcb\ \}) \land \\
\quad\quad (msgtime' = Time)) \\
(BbIdStates = BbsTwoThirdsAttacked) \Rightarrow \\
\quad ((eventname' = McbBbsIdAlarmsOff) \land (eventdestination' = \{\ MyFrb\ \}) \land \\
\quad\quad (eventtime' = Time)) \\
BbIdStates' = BbsThirdAttacked
\end{array}
$$

S — *McbConditionSchemaBbsTwoThirdsAttacked*

$$
\begin{array}{l}
\Delta MoneyCenterBankFSM \\
\Delta OutputEvent \\
\hline
(BbIdStates = BbsThirdAttacked) \land \\
\quad (\#\, BbsIdAlarms \geq (2 * (\#\, MyBbs) \setminus div\ 3)) \\
BbIdStates' = BbsTwoThirdsAttacked \\
eventname' = McbBbsIdAlarmsOn \\
eventdestination' = \{\ MyFrb\ \} \\
eventtime' = Time
\end{array}
$$

S — *McbConditionSchemaBbsAttackedRecovered*

$$
\begin{array}{l}
\Delta MoneyCenterBankFSM \\
\Delta OutputMessage \\
\hline
(BbIdStates = BbsThirdAttacked) \land (\#\, BbsIdAlarms < (\#\, MyBbs \setminus div\ 3)) \\
BbIdStates' = BbsIdOk \\
msgname' = MsgMcbLowerAlert \\
msgdestination' = \{\ MyMcb\ \} \\
msgtime' = Time
\end{array}
$$

10 Branch banks intrusion detection alarms have gone on in the past 60 seconds, then off:

$$
\begin{array}{|l}
\hline
S \quad McbConditionSchema10AlarmsIn60Seconds \\
\Delta MoneyCenterBankFSM \\
\Delta OutputEvent \\
\hline
(BbCoordinatedAttack = False) \land (\#BbsAlarmTimes \geq 10) \land \\
(BbsAlarmTimes\ ((\#BbsAlarmTimes) - 9) \geq (Time - 60)) \\
BbCoordinatedAttack' = True \\
eventname' = McbBbCoordinatedAttack \\
eventdestination' = MyBbs \cup \{ MyFrb \} \\
eventtime' = Time \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
S \quad McbConditionSchema10AlarmsNotIn60Seconds \\
\Delta MoneyCenterBankFSM \\
\Delta OutputEvent \\
\hline
(BbCoordinatedAttack = True) \land \\
(BbsAlarmTimes\ ((\#BbsAlarmTimes) - 9) < (Time - 60)) \\
BbCoordinatedAttack' = False \\
eventname' = McbBbNoCoordinatedAttack \\
eventdestination' = MyBbs \cup \{ MyFrb \} \\
eventtime' = Time \\
\hline
\end{array}
$$

Finite-State Machine File: FederalReserveBankFSM

Initialization Schemas

S *InitFederalReserveBankFSM*

$FederalReserveBankFSM$

$MyFrbs = \varnothing$
$PrimaryFrb = 0$
$MyMcbs = \varnothing$
$NodesDown = FalseFrbUp$
$DbDown = False$
$Alert = False$
$FrbsUp = \varnothing$
$FrbsDown = \varnothing$
$McbsUp = \varnothing$
$McbsDown = \varnothing$
$McbsIdAlarms = \varnothing$
$McbBbsDown = \varnothing$
$McbBbsIdAlarms = \varnothing$
$BbsCorruptDb = \varnothing$
$McbsCorruptDb = \varnothing$
$HalfMcbsIdAlarms = False$

S *InitFrbNewFrb*

$\Delta FederalReserveBankFSM$
$frbid? : \mathbb{N}$

$MyFrbs' = MyFrbs \cup \{ frbid? \}$
$FrbsUp' = FrbsUp \cup \{ frbid? \}$

S *InitFrbPrimaryFrb*

$\Delta FederalReserveBankFSM$
$frbid? : \mathbb{N}$

$PrimaryFrb' = frbid?$

S — *InitFrbNewMcb*

$$\Delta FederalReserveBankFSM$$
$$mcbid? : \mathbb{N}$$

$$MyMcbs' = MyMcbs \cup \{ mcbid? \}$$
$$McbsUp' = McbsUp \cup \{ mcbid? \}$$

Event Schemas (Low-Level/Basic Events)

Local site failure and repair:

S — *FrbSchemaLocalSiteFailure*

$$\Delta FederalReserveBankFSM$$
$$\Delta OutputMessage$$
$$\Delta OutputEvent$$
$$frbdown? : \mathbb{N}$$

$$((NodesDown = FalseFrbUp) \Rightarrow (NodesDown' = Down1Frb)) \lor$$
$$((NodesDown = Down1Frb) \Rightarrow (NodesDown' = Down2Frbs)) \lor$$
$$((NodesDown = Down2Frbs) \Rightarrow$$
$$((NodesDown' = AllFrbsDown) \land (msgname' = MsgFrbShutdown) \land$$
$$(msgdestination' = FrbsUp) \land (msgtime' = Time))))$$
$$FrbsUp' = FrbsUp \setminus \{ frbdown? \}$$
$$FrbsDown' = FrbsDown \cup \{ frbdown? \}$$
$$(((frbdown? = PrimaryFrb) \land ((PrimaryFrb + 1) \in FrbsUp)) \Rightarrow$$
$$((PrimaryFrb' = PrimaryFrb + 1) \land$$
$$(msgname' = MsgFrbSwitchToBackup) \land$$
$$(msgdestination' = FrbsUp) \land (msgtime' = Time))) \lor$$
$$(((frbdown? = PrimaryFrb) \land ((PrimaryFrb + 2) \in FrbsUp)) \Rightarrow$$
$$((PrimaryFrb' = PrimaryFrb + 2) \land$$
$$(msgname' = MsgFrbSwitchToBackup) \land (msgdestination' = FrbsUp) \land$$
$$(msgtime' = Time))))$$
$$eventname' = FrbSiteFailure$$
$$eventdestination' = MyMcbs$$
$$eventtime' = Time$$

S — *FrbSchemaLocalSiteRepair*

$$
\begin{array}{l}
\Delta FederalReserveBankFSM \\
\Delta OutputMessage \\
\Delta OutputEvent \\
frbup? : \mathbb{N} \\
\hline
(((NodesDown = Down2Frbs) \Rightarrow (NodesDown' = Down1Frb)) \lor \\
\quad ((NodesDown = Down1Frb) \Rightarrow (NodesDown' = FalseFrbUp))) \\
FrbsUp' = FrbsUp \cup \{ frbup? \} \\
FrbsDown' = FrbsDown \setminus \{ frbup? \} \\
((PrimaryFrb > frbup?) \Rightarrow \\
\quad ((PrimaryFrb' = frbup?) \land \\
\quad\quad (msgname' = MsgFrbSwitchBackFromBackup) \land \\
\quad\quad (msgdestination' = FrbsUp) \land (msgtime' = Time))) \\
eventname' = FrbSiteRepair \\
eventdestination' = MyMcbs \\
eventtime' = Time
\end{array}
$$

Local database failure and repair:

S — *FrbSchemaLocalDbFailure*

$$
\begin{array}{l}
\Delta FederalReserveBankFSM \\
\Delta OutputMessage \\
\hline
DbDown' = True \\
msgname' = MsgFrbQueueBatches \\
msgdestination' = \{ PrimaryFrb \} \\
msgtime' = Time
\end{array}
$$

S — *FrbSchemaLocalDbRepair*

$$
\begin{array}{l}
\Delta FederalReserveBankFSM \\
\Delta OutputMessage \\
\hline
DbDown' = False \\
msgname' = MsgFrbUnQueueBatches \\
msgdestination' = \{ PrimaryFrb \} \\
msgtime' = Time
\end{array}
$$

Local power failure and repair:

S

FrbSchemaLocalPowerFailure

$$
\begin{array}{l}
\Delta FederalReserveBankFSM \\
\Delta OutputMessage \\
\Delta OutputEvent \\
frbdown? : \mathbb{N} \\
\hline
((NodesDown = FalseFrbUp) \Rightarrow (NodesDown' = Down1Frb)) \lor \\
\quad ((NodesDown = Down1Frb) \Rightarrow (NodesDown' = Down2Frbs)) \lor \\
\quad ((NodesDown = Down2Frbs) \Rightarrow \\
\qquad ((NodesDown' = AllFrbsDown) \land (msgname' = MsgFrbShutdown) \land \\
\qquad\quad (msgdestination' = FrbsUp) \land (msgtime' = Time))) \\
FrbsUp' = FrbsUp \setminus \{ frbdown? \} \\
FrbsDown' = FrbsDown \cup \{ frbdown? \} \\
((((frbdown? = PrimaryFrb) \land ((PrimaryFrb + 1) \in FrbsUp)) \Rightarrow \\
\qquad ((PrimaryFrb' = PrimaryFrb + 1) \land \\
\qquad\quad (msgname' = MsgFrbSwitchToBackup) \land \\
\qquad\quad (msgdestination' = FrbsUp) \land (msgtime' = Time))) \lor \\
\quad (((frbdown? = PrimaryFrb) \land ((PrimaryFrb + 2) \in FrbsUp)) \Rightarrow \\
\qquad ((PrimaryFrb' = PrimaryFrb + 2) \land \\
\qquad\quad (msgname' = MsgFrbSwitchToBackup) \land (msgdestination' = FrbsUp) \land \\
\qquad\quad (msgtime' = Time)))) \\
eventname' = FrbPowerFailure \\
eventdestination' = MyMcbs \\
eventtime' = Time
\end{array}
$$

S *FrbSchemaLocalPowerRepair*

$$
\begin{array}{l}
\Delta FederalReserveBankFSM \\
\Delta OutputMessage \\
\Delta OutputEvent \\
frbup? : \mathbb{N} \\
\hline
(((NodesDown = Down2Frbs) \Rightarrow (NodesDown' = Down1Frb)) \lor \\
\quad ((NodesDown = Down1Frb) \Rightarrow (NodesDown' = FalseFrbUp))) \\
FrbsUp' = FrbsUp \cup \{ frbup? \} \\
FrbsDown' = FrbsDown \setminus \{ frbup? \} \\
((PrimaryFrb > frbup?) \Rightarrow \\
\quad ((PrimaryFrb' = frbup?) \land \\
\quad\quad (msgname' = MsgFrbSwitchBackFromBackup) \land \\
\quad\quad (msgdestination' = FrbsUp) \land (msgtime' = Time))) \\
eventname' = FrbPowerRepair \\
eventdestination' = MyMcbs \\
eventtime' = Time
\end{array}
$$

Local intrusion detection alarm going on and off:

S *FrbSchemaLocalIdAlarmOn*

$$
\begin{array}{l}
\Delta FederalReserveBankFSM \\
\Delta OutputMessage \\
\Delta OutputEvent \\
\hline
Alert' = True \\
msgname' = MsgFrbRaiseAlert \\
msgdestination' = FrbsUp \\
msgtime' = Time \\
eventname' = FrbIdAlarmOn \\
eventdestination' = MyMcbs \\
eventtime' = Time
\end{array}
$$

S *FrbSchemaLocalIdAlarmOff*

$$\Delta FederalReserveBankFSM$$
$$\Delta OutputMessage$$
$$\Delta OutputEvent$$

$$Alert' = False$$
$$msgname' = MsgFrbLowerAlert$$
$$msgdestination' = FrbsUp$$
$$msgtime' = Time$$
$$eventname' = FrbIdAlarmOff$$
$$eventdestination' = MyMcbs$$
$$eventtime' = Time$$

Money Center Bank site failure and repair:

S *FrbSchemaMcbSiteFailure*

$$\Delta FederalReserveBankFSM$$
$$mcbdown? : \mathbb{N}$$

$$McbsDown' = McbsDown \cup \{ mcbdown? \}$$
$$McbsUp' = McbsUp \setminus \{ mcbdown? \}$$
$$McbsUp' = McbsUp \cup \{ (mcbdown? + 1) \}$$

S *FrbSchemaMcbSiteRepair*

$$\Delta FederalReserveBankFSM$$
$$mcbup? : \mathbb{N}$$

$$McbsDown' = McbsDown \setminus \{ mcbup? \}$$
$$McbsUp' = McbsUp \cup \{ mcbup? \}$$
$$McbsUp' = McbsUp \setminus \{ (mcbup? + 1) \}$$

Money Center Bank power failure and repair:

S *FrbSchemaMcbPowerFailure*

$$\Delta FederalReserveBankFSM$$
$$mcbdown? : \mathbb{N}$$

$$McbsDown' = McbsDown \cup \{ mcbdown? \}$$
$$McbsUp' = McbsUp \setminus \{ mcbdown? \}$$
$$McbsUp' = McbsUp \cup \{ (mcbdown? + 1) \}$$

$$
\begin{array}{|l}
\hline
\mathbf{S}\ \ FrbSchemaMcbPowerRepair \\
\Delta FederalReserveBankFSM \\
mcbup? : \mathbb{N} \\
\hline
McbsDown' = McbsDown \setminus \{ mcbup? \} \\
McbsUp' = McbsUp \cup \{ mcbup? \} \\
McbsUp' = McbsUp \setminus \{ (mcbup? + 1) \} \\
\hline
\end{array}
$$

Money Center Bank intrusion detection alarm going on and off:

$$
\begin{array}{|l}
\hline
\mathbf{S}\ \ FrbSchemaMcbIdAlarmOn \\
\Delta FederalReserveBankFSM \\
mcbidalarmon? : \mathbb{N} \\
\hline
McbsIdAlarms' = McbsIdAlarms \cup \{ mcbidalarmon? \} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathbf{S}\ \ FrbSchemaMcbIdAlarmOff \\
\Delta FederalReserveBankFSM \\
mcbidalarmoff? : \mathbb{N} \\
\hline
McbsIdAlarms' = McbsIdAlarms \setminus \{ mcbidalarmoff? \} \\
\hline
\end{array}
$$

Branch Bank database failure and repair:

$$
\begin{array}{|l}
\hline
\mathbf{S}\ \ FrbSchemaMcbBbDbFailure \\
\Delta FederalReserveBankFSM \\
bbdbdown? : \mathbb{N} \\
\hline
BbsCorruptDb' = BbsCorruptDb \cup \{ bbdbdown? \} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathbf{S}\ \ FrbSchemaMcbBbDbRepair \\
\Delta FederalReserveBankFSM \\
bbdbup? : \mathbb{N} \\
\hline
BbsCorruptDb' = BbsCorruptDb \setminus \{ bbdbup? \} \\
\hline
\end{array}
$$

Money Center Bank database failure and repair:

S — *FrbSchemaMcbDbFailure*

$$\Delta FederalReserveBankFSM$$
$$mcbdbdown? : \mathbb{N}$$
$$\overline{McbsCorruptDb' = McbsCorruptDb \cup \{ mcbdbdown? \}}$$

S — *FrbSchemaMcbDbRepair*

$$\Delta FederalReserveBankFSM$$
$$mcbdbup? : \mathbb{N}$$
$$\overline{McbsCorruptDb' = McbsCorruptDb \setminus \{ mcbdbup? \}}$$

High-Level Events

Half of the Money Center Banks have failed:

S — *FrbConditionSchemaHalfMcbsDown*

$$\Delta FederalReserveBankFSM$$
$$\Delta OutputMessage$$
$$\Delta OutputEvent$$
$$\overline{(\#McbsDown \geq (\#MyMcbs \setminus div\ 2))}$$
$$msgname' = MsgFrbShutdown$$
$$msgdestination' = MyFrbs$$
$$msgtime' = Time$$
$$eventname' = SystemShutdown$$
$$eventdestination' = MyMcbs$$
$$eventtime' = Time$$

Half of the Money Center Banks have intrusion detection alarms going on and off:

S — *FrbConditionSchemaHalfMcbsIdAlarmsOn*

$$
\begin{array}{l}
\Delta FederalReserveBankFSM \\
\Delta OutputMessage \\
\Delta OutputEvent \\
\hline
(\#\, McbsIdAlarms \geq (\#\, MyMcbs \setminus div\ 2)) \wedge (HalfMcbsIdAlarms = False) \\
HalfMcbsIdAlarms' = True \\
msgname' = MsgFrbRaiseAlert \\
msgdestination' = MyFrbs \\
msgtime' = Time \\
eventname' = FrbIdAlarmOn \\
eventdestination' = MyMcbs \\
eventtime' = Time
\end{array}
$$

S — *FrbConditionSchemaHalfMcbIdAlarmsOff*

$$
\begin{array}{l}
\Delta FederalReserveBankFSM \\
\Delta OutputMessage \\
\Delta OutputEvent \\
\hline
(\#\, McbsIdAlarms < (\#\, MyMcbs \setminus div\ 2)) \wedge (HalfMcbsIdAlarms = True) \\
HalfMcbsIdAlarms' = False \\
msgname' = MsgFrbLowerAlert \\
msgdestination' = MyFrbs \\
msgtime' = Time \\
eventname' = FrbIdAlarmOff \\
eventdestination' = MyMcbs \\
eventtime' = Time
\end{array}
$$

Half of the Money Center Banks have two-thirds of their Branch Banks down:

$$
\begin{array}{|l}
\hline
S \quad FrbConditionSchemaHalfMcbBbsDown \\
\Delta FederalReserveBankFSM \\
\Delta OutputMessage \\
\Delta OutputEvent \\
\hline
(\#\, McbBbsDown \geq (\#\, MyMcbs \setminus div\ 2)) \\
msgname' = MsgFrbShutdown \\
msgdestination' = MyFrbs \\
msgtime' = Time \\
eventname' = SystemShutdown \\
eventdestination' = MyMcbs \\
eventtime' = Time \\
\hline
\end{array}
$$

Half of the Money Center Banks have two-thirds of their Branch Banks intrusion detections alarms on:

$$
\begin{array}{|l}
\hline
S \quad FrbConditionSchemaHalfMcbBbsIdAlarms \\
\Delta FederalReserveBankFSM \\
\Delta OutputMessage \\
\Delta OutputEvent \\
\hline
(\#\, McbBbsIdAlarms \geq (\#\, MyMcbs \setminus div\ 2)) \\
msgname' = MsgFrbShutdown \\
msgdestination' = MyFrbs \\
msgtime' = Time \\
eventname' = SystemShutdown \\
eventdestination' = MyMcbs \\
eventtime' = Time \\
\hline
\end{array}
$$

C.3 Example Specification: Electric Power System

The next two subsections comprise the RAPTOR specification of fault tolerance for an electric power application model.

C.3.1 System Specification

This subsection provides the class definitions of the System Specification for the electric power model.

PowerCompany Class

```
// Class definition
persistent class PowerCompany
{
 public: // Constructors & destructor
      PowerCompany(const int pcId, const PtString& name);
      virtual ~PowerCompany();

 public: // Services
      void Print() const;

 public: // Accessors
      const int pcId() const;
      const PtString& Name(void) const;
      void SetName(const PtString& name);

 private: // Attributes
      int m_pcId;
      PtString m_name;
      HardwarePlatform m_HardwarePlatform;
      OperatingSystem m_OperatingSystem;

 public: // Pointers
      ControlArea* m_controlArea;

 public:
      bool NewObject;
      PowerCompanyFSM m_fsm;

 private: // Index
      useindex PcIdIndex;
};


// index definition
indexdef PcIdIndex : PowerCompany
{
```

```
      m_pcId;
};

// set definition
typedef lset<PowerCompany*> PowerCompanySet;
```

ControlArea Class

```
// Class definition
persistent class ControlArea
{
 public: // Constructors & destructor
      ControlArea(const int caId, const PtString& name);
      virtual ~ControlArea();

 public: // Services
      void Print() const;

 public: // Accessors
      const int caId() const;
      const PtString& Name(void) const;
      void SetName(const PtString& name);

 private: // Attributes
      int m_caId;
      PtString m_name;
      HardwarePlatform m_HardwarePlatform;
      OperatingSystem m_OperatingSystem;

 public: // Pointers
      ControlRegion* m_controlRegion;

 public: // Set
      lset<PowerCompany*> m_powerCompanies;

 public:
      bool NewObject;
      ControlAreaFSM m_fsm;

 private: // Index
      useindex CaIdIndex;
};


// index definition
indexdef CaIdIndex : ControlArea
{
      m_caId;
};
```

```
// set definition
typedef lset<ControlArea*> ControlAreaSet;
```

ControlRegion Class

```
// Class definition
persistent class ControlRegion
{
 public: // Constructors & destructor
      ControlRegion(const int crId, const PtString& name);
      virtual ~ControlRegion();

 public: // Services
      void Print() const;

 public: // Accessors
      const int crId() const;
      const PtString& Name(void) const;
      void SetName(const PtString& name);

 private: // Attributes
      int m_crId;
      PtString m_name;
      HardwarePlatform m_HardwarePlatform;
      OperatingSystem m_OperatingSystem;

 public: // Set
      lset<ControlArea*> m_controlAreas;

 public:
      bool NewObject;
      ControlRegionFSM m_fsm;

 private: // Index
      useindex CrIdIndex;
};


// index definition
indexdef CrIdIndex : ControlRegion
{
      m_crId;
};

// set definition
typedef lset<ControlRegion*> ControlRegionSet;
```

C.3.2 Error Detection and Recovery Specifications

This subsection contains the fault-tolerance specifications for the power model, organized according to file type.

Declarations File

Given Sets

Z
$$[Generator, Substation, PowerCompany, ControlArea, ControlRegion, Interconnection]$$

Axiomatic Descriptions

A
$$Time : \mathbb{N}$$
$$Time > 0$$

Set Definitions

Z
$$bool ::= True \mid False$$

Z
$$\begin{aligned} SystemEvents ::= \; & NullEvent \\ \mid \; & LocalSiteFailure \\ \mid \; & LocalSiteRepair \\ \mid \; & LocalDbFailure \\ \mid \; & LocalDbRepair \\ \mid \; & LocalIdAlarmOn \\ \mid \; & LocalIdAlarmOff \end{aligned}$$

Z

| *PcSiteFailure*
| *PcSiteRepair*
| *PcIdAlarmOn*
| *PcIdAlarmOff*
| *GenSiteFailure*
| *GenSiteRepair*
| *SubSiteFailure*
| *SubSiteRepair*
| *GenIdAlarmOn*
| *GenIdAlarmOff*
| *SubIdAlarmOn*
| *SubIdAlarmOff*
| *CaSiteFailure*
| *CaSiteRepair*
| *CaIdAlarmOn*
| *CaIdAlarmOff*
| *PcTwoThirdsGensDown*
| *PcLessTwoThirdsGensDown*
| *PcAllGensIdAlarms*
| *PcNotAllGensIdAlarms*
| *PcHalfSubsIdAlarms*
| *PcLessHalfSubsIdAlarms*
| *CaPcTwoThirdsGensDown*
| *CaPcLessTwoThirdsGensDown*
| *CrSiteFailure*
| *CrSiteRepair*
| *CrIdAlarmOn*
| *CrIdAlarmOff*
| *CaTwoThirdsPcIdAlarms*
| *CaLessTwoThirdsPcIdAlarms*
| *CaTwoThirdsPcsWithGensIdAlarms*
| *CaLessTwoThirdsPcsWithGensIdAlarms*
| *CaAllPcsWithSubsIdAlarms*
| *CaNotAllPcsWithSubsIdAlarms*
| *NipcCrCoordinatedAttack*
| *NipcNotCrCoordinatedAttack*
| *NipcCaCoordinatedAttack*
| *NipcNotCaCoordinatedAttack*
| *NipcPcCoordinatedAttack*
| *NipcNotPcCoordinatedAttack*

Z

$$
\begin{aligned}
MessagesToApplication ::= {} & NullMessage \\
| \; & MsgPcSwitchToBackupPc \\
| \; & MsgPcSwitchBackFromBackupPc \\
| \; & MsgPcLocalBalancing \\
| \; & MsgPcNormalBalancing \\
| \; & MsgPcRaiseAlert \\
| \; & MsgPcLowerAlert \\
| \; & MsgPcGenMorePower \\
| \; & MsgPcGenLessPower \\
| \; & MsgPcSwitchToBackupCa \\
| \; & MsgPcSwitchBackFromBackupCa \\
| \; & MsgCaLocalBalancing \\
| \; & MsgCaNormalBalancing \\
| \; & MsgCaRaiseAlert \\
| \; & MsgCaLowerAlert \\
| \; & MsgCaSwitchToBackupPc \\
| \; & MsgCaSwitchBackFromBackupPc \\
| \; & MsgCaSwitchToBackupCr \\
| \; & MsgCaSwitchBackFromBackupCr \\
| \; & MsgCrLocalBalancing \\
| \; & MsgCrNormalBalancing \\
| \; & MsgCrRaiseAlert \\
| \; & MsgCrLowerAlert \\
| \; & MsgCrSwitchToBackupCa \\
| \; & MsgCrSwitchBackFromBackupCa
\end{aligned}
$$

Schema Structures

S

OutputEvent

$$
\begin{aligned}
& eventname : SystemEvents \\
& eventdestination : \mathbb{F}\, \mathbb{N} \\
& eventtime : \mathbb{N} \\
\hline
& eventname \neq NullEvent \\
& eventdestination \neq \varnothing \\
& eventtime \neq 0
\end{aligned}
$$

S — *OutputMessage*

$$
\begin{array}{l}
msgname : MessagesToApplication \\
msgdestination : \mathbb{F}\ \mathbb{N} \\
msgtime : \mathbb{N} \\
\hline
msgdestination \neq \varnothing \\
msgtime \neq 0
\end{array}
$$

State Description File

State Schemas

S — *PowerCompanyFSM*

$MyPc : \mathbb{N}$
$MyControlArea : \mathbb{N}$
$MyGens : \mathbb{F}\ \mathbb{N}$
$MySubs : \mathbb{F}\ \mathbb{N}$
$GensUp : \mathbb{F}\ \mathbb{N}$
$GensDown : \mathbb{F}\ \mathbb{N}$
$SubsUp : \mathbb{F}\ \mathbb{N}$
$SubsDown : \mathbb{F}\ \mathbb{N}$
$NodeDown : bool$
$DbDown : bool$
$IdAlarm : bool$
$GenIdAlarms : \mathbb{F}\ \mathbb{N}$
$SubIdAlarms : \mathbb{F}\ \mathbb{N}$
$SubIdAlarmTimes : seq\ \mathbb{N}$
$AlertAllFromCa : bool$
$TwoThirdsGensDown : bool$
$AllGensIdAlarms : bool$
$HalfSubsIdAlarmsIn60Seconds : bool$

$GensUp \subseteq MyGens$
$GensDown \subseteq MyGens$
$GensUp \cup GensDown = MyGens$
$SubsUp \subseteq MySubs$
$SubsDown \subseteq MySubs$
$SubsUp \cup SubsDown = MySubs$
$GenIdAlarms \subseteq MyGens$
$SubIdAlarms \subseteq MySubs$

S *ControlAreaFSM*

$$
\begin{array}{l}
MyCa : \mathbb{N} \\
MyControlRegion : \mathbb{N} \\
Nipc : \mathbb{N} \\
MyPcs : \mathbb{F}\ \mathbb{N} \\
PcsUp : \mathbb{F}\ \mathbb{N} \\
PcsDown : \mathbb{F}\ \mathbb{N} \\
NodeDown : bool \\
DbDown : bool \\
IdAlarm : bool \\
PcIdAlarms : \mathbb{F}\ \mathbb{N} \\
AlertAllFromCr : bool \\
PcsWithGensDown : \mathbb{F}\ \mathbb{N} \\
PcsWithGensIdAlarms : \mathbb{F}\ \mathbb{N} \\
PcsWithSubsIdAlarms : \mathbb{F}\ \mathbb{N} \\
TwoThirdsPcIdAlarms : bool \\
TwoThirdsPcsWithGensIdAlarms : bool \\
AllPcsWithSubsIdAlarms : bool \\
\hline
PcsUp \subseteq MyPcs \\
PcsDown \subseteq MyPcs \\
PcsUp \cup PcsDown = MyPcs \\
PcIdAlarms \subseteq MyPcs
\end{array}
$$

S *ControlRegionFSM*

$$
\begin{array}{l}
MyCr : \mathbb{N} \\
Nipc : \mathbb{N} \\
MyCas : \mathbb{F}\ \mathbb{N} \\
CasUp : \mathbb{F}\ \mathbb{N} \\
CasDown : \mathbb{F}\ \mathbb{N} \\
NodeDown : bool \\
DbDown : bool \\
IdAlarm : bool \\
\hline
CasUp \subseteq MyCas \\
CasDown \subseteq MyCas \\
CasUp \cup CasDown = MyCas
\end{array}
$$

S — *NipcFSM*

$$
\begin{array}{l}
MyNipc : \mathbb{N} \\
AllCrs : \mathbb{F}\ \mathbb{N} \\
AllCas : \mathbb{F}\ \mathbb{N} \\
CrsIdAlarms : \mathbb{F}\ \mathbb{N} \\
CasIdAlarms : \mathbb{F}\ \mathbb{N} \\
CasPcsIdAlarms : \mathbb{F}\ \mathbb{N} \\
CasGensIdAlarms : \mathbb{F}\ \mathbb{N} \\
CasSubsIdAlarms : \mathbb{F}\ \mathbb{N} \\
CrCoordinatedAttack : bool \\
CaCoordinatedAttack : bool \\
PcCoordinatedAttack : bool \\
\hline
CrsIdAlarms \subseteq AllCrs \\
CasIdAlarms \subseteq AllCas \\
CasPcsIdAlarms \subseteq AllCas \\
CasGensIdAlarms \subseteq AllCas \\
CasSubsIdAlarms \subseteq AllCas
\end{array}
$$

System State Schema

S — *PowerSystem*

$$
\begin{array}{l}
AllPowerCompanies : \mathbb{P}\ PowerCompanyFSM \\
AllControlAreas : \mathbb{P}\ ControlAreaFSM \\
AllControlRegions : \mathbb{P}\ ControlRegionFSM \\
Nipc : NipcFSM \\
\hline
\#\ AllPowerCompanies > \#\ AllControlAreas
\end{array}
$$

Finite-State Machine File: PowerCompanyFSM

Initialization Schemas

S *InitPowerCompanyFSM*

$PowerCompanyFSM$

$MyPc = 0$
$MyControlArea = 0$
$MyGens = \varnothing$
$MySubs = \varnothing$
$GensUp = \varnothing$
$GensDown = \varnothing$
$SubsUp = \varnothing$
$SubsDown = \varnothing$
$NodeDown = False$
$DbDown = False$
$IdAlarm = False$
$GenIdAlarms = \varnothing$
$SubIdAlarms = \varnothing$
$SubIdAlarmTimes = \langle\ \rangle$
$AlertAllFromCa = False$
$TwoThirdsGensDown = False$
$AllGensIdAlarms = False$
$HalfSubsIdAlarmsIn60Seconds = False$

S *InitPcMyPc*

$\Delta PowerCompanyFSM$
$pcid? : \mathbb{N}$

$MyPc' = pcid?$

S *InitPcMyControlArea*

$\Delta PowerCompanyFSM$
$caid? : \mathbb{N}$

$MyControlArea' = caid?$

S *InitPcNewGen*

$\Delta PowerCompanyFSM$
$genid? : \mathbb{N}$

$MyGens' = MyGens \cup \{ genid? \}$
$GensUp' = GensUp \cup \{ genid? \}$

S *InitPcNewSub*

$\Delta PowerCompanyFSM$
$subid? : \mathbb{N}$

$MySubs' = MySubs \cup \{ subid? \}$
$SubsUp' = SubsUp \cup \{ subid? \}$

Event Schemas (Low-Level/Basic Events)

Local site failure and repair:

S *PcSchemaLocalSiteFailure*

$\Delta PowerCompanyFSM$
$\Delta OutputEvent$
$\Delta OutputMessage$

$NodeDown' = True$
$eventname' = PcSiteFailure$
$eventdestination' = \{ MyControlArea \}$
$eventtime' = Time$
$msgname' = MsgPcSwitchToBackupPc$
$msgdestination' = MyGens \cup MySubs$
$msgtime' = Time$

S — *PcSchemaLocalSiteRepair*

$\Delta PowerCompanyFSM$
$\Delta OutputEvent$
$\Delta OutputMessage$

$NodeDown' = False$
$eventname' = PcSiteRepair$
$eventdestination' = \{ MyControlArea \}$
$eventtime' = Time$
$msgname' = MsgPcSwitchBackFromBackupPc$
$msgdestination' = MyGens \cup MySubs$
$msgtime' = Time$

Local database failure and repair:

S — *PcSchemaLocalDbFailure*

$\Delta PowerCompanyFSM$
$\Delta OutputMessage$

$DbDown' = True$
$msgname' = MsgPcLocalBalancing$
$msgdestination' = \{ MyPc \}$
$msgtime' = Time$

S — *PcSchemaLocalDbRepair*

$\Delta PowerCompanyFSM$
$\Delta OutputMessage$

$DbDown' = False$
$msgname' = MsgPcNormalBalancing$
$msgdestination' = \{ MyPc \}$
$msgtime' = Time$

Local intrustion detection alarm going on and off:

S — *PcSchemaLocalIdAlarmOn*

$\Delta PowerCompanyFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$

$IdAlarm' = True$
$msgname' = MsgPcRaiseAlert$
$msgdestination' = \{ MyPc \}$
$msgtime' = Time$
$eventname' = PcIdAlarmOn$
$eventdestination' = \{ MyControlArea \}$
$eventtime' = Time$

S — *PcSchemaLocalIdAlarmOff*

$\Delta PowerCompanyFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$
$bbidalarmoff? : \mathbb{N}$

$IdAlarm' = False$
$msgname' = MsgPcLowerAlert$
$msgdestination' = \{ MyPc \}$
$msgtime' = Time$
$eventname' = PcIdAlarmOff$
$eventdestination' = \{ MyControlArea \}$
$eventtime' = Time$

Generator site failure and repair:

S — *PcSchemaGenSiteFailure*

$\Delta PowerCompanyFSM$
$\Delta OutputMessage$
$genid? : \mathbb{N}$

$GensDown' = GensDown \cup \{ genid? \}$
$GensUp' = GensUp \setminus \{ genid? \}$
$msgname' = MsgPcGenMorePower$
$msgdestination' = GensUp'$
$msgtime' = Time$

S — *PcSchemaGenSiteRepair*

$\Delta PowerCompanyFSM$
$\Delta OutputMessage$
$genid? : \mathbb{N}$

$msgname' = MsgPcGenLessPower$
$msgdestination' = GensUp$
$msgtime' = Time$
$GensUp' = GensUp \cup \{ genid? \}$
$GensDown' = GensDown \setminus \{ genid? \}$

Substation site failure and repair:

S — *PcSchemaSubSiteFailure*

$\Delta PowerCompanyFSM$
$subid? : \mathbb{N}$

$SubsDown' = SubsDown \cup \{ subid? \}$
$SubsUp' = SubsUp \setminus \{ subid? \}$

S — *PcSchemaSubSiteRepair*

$\Delta PowerCompanyFSM$
$subid? : \mathbb{N}$

$SubsUp' = SubsUp \cup \{ subid? \}$
$SubsDown' = SubsDown \setminus \{ subid? \}$

Generator intrusion detection alarm going on and off:

S — *PcSchemaGenIdAlarmOn*

$\Delta PowerCompanyFSM$
$genid? : \mathbb{N}$

$GenIdAlarms' = GenIdAlarms \cup \{ genid? \}$

S — *PcSchemaGenIdAlarmOff*

$\Delta PowerCompanyFSM$
$genid? : \mathbb{N}$

$GenIdAlarms' = GenIdAlarms \setminus \{ genid? \}$

Substation intrusion detection alarm going on and off:

S — *PcSchemaSubIdAlarmOn*

$\Delta PowerCompanyFSM$
$subid? : \mathbb{N}$

$SubIdAlarms' = SubIdAlarms \setminus \{ subid? \}$
$SubIdAlarmTimes' = SubIdAlarmTimes \frown \langle Time \rangle$

S — *PcSchemaSubIdAlarmOff*

$\Delta PowerCompanyFSM$
$subid? : \mathbb{N}$

$SubIdAlarms' = SubIdAlarms \setminus \{ subid? \}$

Control Area site failure and repair:

S — *PcSchemaCaSiteFailure*

$\Delta PowerCompanyFSM$
$\Delta OutputMessage$

$msgname' = MsgPcSwitchToBackupCa$
$msgdestination' = \{ MyPc \}$
$msgtime' = Time$

S — *PcSchemaCaSiteRepair*

$\Delta PowerCompanyFSM$
$\Delta OutputMessage$

$msgname' = MsgPcSwitchBackFromBackupCa$
$msgdestination' = \{ MyPc \}$
$msgtime' = Time$

Control Area intrustion detection alarm going on and off:

S — *PcSchemaCaIdAlarmOn*

$\Delta PowerCompanyFSM$
$\Delta OutputMessage$

$AlertAllFromCa' = True$
$msgname' = MsgPcRaiseAlert$
$msgdestination' = \{ MyPc \}$
$msgtime' = Time$

S — *PcSchemaCaIdAlarmOff*

$\Delta PowerCompanyFSM$
$\Delta OutputMessage$

$AlertAllFromCa' = False$
$msgname' = MsgPcLowerAlert$
$msgdestination' = \{ MyPc \}$
$msgtime' = Time$

Control area's power company has two-thirds of its generators failed and recovered:

S — *PcSchemaCaPcTwoThirdsGensDown*

$\Delta PowerCompanyFSM$
$\Delta OutputMessage$

$msgname' = MsgPcGenMorePower$
$msgdestination' = MyGens$
$msgtime' = Time$

S — *PcSchemaCaPcLessTwoThirdsGensDown*

$\Delta PowerCompanyFSM$
$\Delta OutputMessage$

$msgname' = MsgPcGenLessPower$
$msgdestination' = MyGens$
$msgtime' = Time$

Event Schemas (High-level Events)

Two thirds of the generators for this power company have failed and recovered:

S — *PcConditionSchemaTwoThirdsGensDown*

$\Delta PowerCompanyFSM$
$\Delta OutputEvent$

$(\#\, GensDown = 2) \land (TwoThirdsGensDown = False)$
$TwoThirdsGensDown' = True$
$eventname' = PcTwoThirdsGensDown$
$eventdestination' = \{\, MyControlArea \,\}$
$eventtime' = Time$

S — *PcConditionSchemaLessTwoThirdsGensDown*

$\Delta PowerCompanyFSM$
$\Delta OutputEvent$

$(\#\, GensDown < 2) \land (TwoThirdsGensDown = True)$
$TwoThirdsGensDown' = False$
$eventname' = PcLessTwoThirdsGensDown$
$eventdestination' = \{\, MyControlArea \,\}$
$eventtime' = Time$

All the generators for this power company have had intrusion detection alarms going on and off:

S — *PcConditionSchemaAllGensIdAlarms*

$\Delta PowerCompanyFSM$
$\Delta OutputEvent$

$(\#\, GenIdAlarms = \#\, MyGens)$
$AllGensIdAlarms' = True$
$eventname' = PcAllGensIdAlarms$
$eventdestination' = \{\, MyControlArea \,\}$
$eventtime' = Time$

S — *PcConditionSchemaNotAllGensIdAlarms*

$\Delta PowerCompanyFSM$
$\Delta OutputEvent$

$(\#\, GenIdAlarms \neq \#\, MyGens) \wedge (AllGensIdAlarms = True)$
$AllGensIdAlarms' = False$
$eventname' = PcNotAllGensIdAlarms$
$eventdestination' = \{\ MyControlArea\ \}$
$eventtime' = Time$

Half the substations for this power company have had intrusion detection alarms go on and off in the past 60 seconds:

S — *PcConditionSchemaHalfSubsIdAlarmsIn60Seconds*

$\Delta PowerCompanyFSM$
$\Delta OutputEvent$

$(SubIdAlarmTimes\ ((\#\, SubIdAlarmTimes) - 1) \geq (Time - 60)) \wedge$
$(HalfSubsIdAlarmsIn60Seconds = False) \wedge (\ (\#\, SubIdAlarmTimes) \geq 2)$
$HalfSubsIdAlarmsIn60Seconds' = True$
$eventname' = PcHalfSubsIdAlarms$
$eventdestination' = \{\ MyControlArea\ \}$
$eventtime' = Time$

S — *PcConditionSchemaLessHalfSubsIdAlarmsIn60Seconds*

$\Delta PowerCompanyFSM$
$\Delta OutputEvent$

$(SubIdAlarmTimes\ ((\#\, SubIdAlarmTimes) - 1) < (Time - 60)) \wedge$
$(HalfSubsIdAlarmsIn60Seconds = True)$
$HalfSubsIdAlarmsIn60Seconds' = False$
$eventname' = PcLessHalfSubsIdAlarms$
$eventdestination' = \{\ MyControlArea\ \}$
$eventtime' = Time$

Finite-State Machine File: ControlAreaFSM

Initialization Schemas

S *InitControlAreaFSM*

$ControlAreaFSM$

$MyCa = 0$
$MyControlRegion = 0$
$Nipc = 0$
$MyPcs = \varnothing$
$PcsUp = \varnothing$
$PcsDown = \varnothing$
$NodeDown = False$
$DbDown = False$
$IdAlarm = False$
$PcIdAlarms = \varnothing$
$AlertAllFromCr = False$
$PcsWithGensDown = \varnothing$
$PcsWithGensIdAlarms = \varnothing$
$PcsWithSubsIdAlarms = \varnothing$
$TwoThirdsPcIdAlarms = False$
$TwoThirdsPcsWithGensIdAlarms = False$
$AllPcsWithSubsIdAlarms = False$

S *InitCaMyCa*

$\Delta ControlAreaFSM$
$caid? : \mathbb{N}$

$MyCa' = caid?$

S *InitCaMyCr*

$\Delta ControlAreaFSM$
$crid? : \mathbb{N}$

$MyControlRegion' = crid?$

S *InitCaNipc*

$\Delta ControlAreaFSM$
$nipcid? : \mathbb{N}$

$Nipc' = nipcid?$

S *InitCaNewPc*

$\Delta ControlAreaFSM$
$pcid? : \mathbb{N}$

$MyPcs' = MyPcs \cup \{ pcid? \}$
$PcsUp' = PcsUp \cup \{ pcid? \}$

Event Schemas (Low-Level/Basic Events)

Local site failure and repair:

S *CaSchemaLocalSiteFailure*

$\Delta ControlAreaFSM$
$\Delta OutputEvent$

$NodeDown' = True$
$eventname' = CaSiteFailure$
$eventdestination' = MyPcs \cup \{ MyControlRegion \}$
$eventtime' = Time$

S *CaSchemaLocalSiteRepair*

$\Delta ControlAreaFSM$
$\Delta OutputEvent$

$NodeDown' = False$
$eventname' = CaSiteRepair$
$eventdestination' = MyPcs \cup \{ MyControlRegion \}$
$eventtime' = Time$

Local database failure and repair:

S — *CaSchemaLocalDbFailure*

$\Delta ControlAreaFSM$
$\Delta OutputMessage$

$DbDown' = True$
$msgname' = MsgCaLocalBalancing$
$msgdestination' = \{ MyCa \}$
$msgtime' = Time$

S — *CaSchemaLocalDbRepair*

$\Delta ControlAreaFSM$
$\Delta OutputMessage$

$DbDown' = False$
$msgname' = MsgCaNormalBalancing$
$msgdestination' = \{ MyCa \}$
$msgtime' = Time$

Local intrusion detection alarm going on and off:

S — *CaSchemaLocalIdAlarmOn*

$\Delta ControlAreaFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$

$IdAlarm' = True$
$msgname' = MsgCaRaiseAlert$
$msgdestination' = \{ MyCa \}$
$msgtime' = Time$
$eventname' = CaIdAlarmOn$
$eventdestination' = MyPcs \cup \{ Nipc \}$
$eventtime' = Time$

S *CaSchemaLocalIdAlarmOff*

$\Delta ControlAreaFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$

$IdAlarm' = False$
$msgname' = MsgCaLowerAlert$
$msgdestination' = \{ MyCa \}$
$msgtime' = Time$
$eventname' = CaIdAlarmOff$
$eventdestination' = MyPcs \cup \{ Nipc \}$
$eventtime' = Time$

Power company site failure and repair:

S *CaSchemaPcSiteFailure*

$\Delta ControlAreaFSM$
$\Delta OutputMessage$
$pcid? : \mathbb{N}$

$PcsDown' = PcsDown \cup \{ pcid? \}$
$PcsUp' = PcsUp \setminus \{ pcid? \}$
$msgname' = MsgCaSwitchToBackupPc$
$msgdestination' = \{ MyCa \}$
$msgtime' = Time$

S *CaSchemaPcSiteRepair*

$\Delta ControlAreaFSM$
$\Delta OutputMessage$
$pcid? : \mathbb{N}$

$PcsUp' = PcsUp \cup \{ pcid? \}$
$PcsDown' = PcsDown \setminus \{ pcid? \}$
$msgname' = MsgCaSwitchBackFromBackupPc$
$msgdestination' = \{ MyCa \}$
$msgtime' = Time$

Power company intrusion detection alarm going on and off:

S — *CaSchemaPcIdAlarmOn*

$$\Delta ControlAreaFSM$$
$$pcid? : \mathbb{N}$$
$$PcIdAlarms' = PcIdAlarms \cup \{ pcid? \}$$

S — *CaSchemaPcIdAlarmOff*

$$\Delta ControlAreaFSM$$
$$pcid? : \mathbb{N}$$
$$PcIdAlarms' = PcIdAlarms \setminus \{ pcid? \}$$

Power company with two-thirds of its generators down and recovered:

S — *CaSchemaPcTwoThirdsGensDown*

$$\Delta ControlAreaFSM$$
$$\Delta OutputEvent$$
$$pcid? : \mathbb{N}$$
$$PcsWithGensDown' = PcsWithGensDown \cup \{ pcid? \}$$
$$eventname' = CaPcTwoThirdsGensDown$$
$$eventdestination' = MyPcs \setminus PcsWithGensDown'$$
$$eventtime' = Time$$

S — *CaSchemaPcLessTwoThirdsGensDown*

$$\Delta ControlAreaFSM$$
$$\Delta OutputEvent$$
$$pcid? : \mathbb{N}$$
$$eventname' = CaPcLessTwoThirdsGensDown$$
$$eventdestination' = MyPcs \setminus PcsWithGensDown$$
$$eventtime' = Time$$
$$PcsWithGensDown' = PcsWithGensDown \setminus \{ pcid? \}$$

Power company with all its generators' intrusion detection alarms going on and off:

S — *CaSchemaPcAllGensIdAlarms*

$$\Delta ControlAreaFSM$$
$$pcid? : \mathbb{N}$$
$$PcsWithGensIdAlarms' = PcsWithGensIdAlarms \cup \{ pcid? \}$$

S — *CaSchemaPcNotAllGensIdAlarms*

$$\Delta ControlAreaFSM$$
$$pcid? : \mathbb{N}$$
$$PcsWithGensIdAlarms' = PcsWithGensIdAlarms \setminus \{ pcid? \}$$

Power company with half its substations' intrusion detection alarms going on and off:

S — *CaSchemaPcHalfSubsIdAlarms*

$$\Delta ControlAreaFSM$$
$$pcid? : \mathbb{N}$$
$$PcsWithSubsIdAlarms' = PcsWithSubsIdAlarms \cup \{ pcid? \}$$

S — *CaSchemaPcLessHalfSubsIdAlarms*

$$\Delta ControlAreaFSM$$
$$pcid? : \mathbb{N}$$
$$PcsWithSubsIdAlarms' = PcsWithSubsIdAlarms \setminus \{ pcid? \}$$

Control region site failure and repair:

S — *CaSchemaCrSiteFailure*

$$\Delta ControlAreaFSM$$
$$\Delta OutputMessage$$
$$msgname' = MsgCaSwitchToBackupCr$$
$$msgdestination' = \{ MyCa \}$$
$$msgtime' = Time$$

S *CaSchemaCrSiteRepair*

$\Delta ControlAreaFSM$
$\Delta OutputMessage$

$msgname' = MsgCaSwitchBackFromBackupCr$
$msgdestination' = \{ MyCa \}$
$msgtime' = Time$

Control region intrusion detection alarm going on and off:

S *CaSchemaCrIdAlarmOn*

$\Delta ControlAreaFSM$
$\Delta OutputMessage$

$AlertAllFromCr' = True$
$msgname' = MsgCaRaiseAlert$
$msgdestination' = \{ MyCa \}$
$msgtime' = Time$

S *CaSchemaCrIdAlarmOff*

$\Delta ControlAreaFSM$
$\Delta OutputMessage$

$AlertAllFromCr' = False$
$msgname' = MsgCaLowerAlert$
$msgdestination' = \{ MyCa \}$
$msgtime' = Time$

Event Schemas (High-level Events)

Two-thirds of the power companies have intrusion detection alarms going on and off:

S *CaConditionSchemaTwoThirdsPcIdAlarms*

$\Delta ControlAreaFSM$
$\Delta OutputEvent$

$(\#PcIdAlarms = 2) \land (TwoThirdsPcIdAlarms = False)$
$TwoThirdsPcIdAlarms' = True$
$eventname' = CaTwoThirdsPcIdAlarms$
$eventdestination' = \{ Nipc \}$
$eventtime' = Time$

S — *CaConditionSchemaLessTwoThirdsPcIdAlarms*

$\Delta ControlAreaFSM$
$\Delta OutputEvent$

$(\#\, PcIdAlarms < 2) \land (TwoThirdsPcIdAlarms = True)$
$TwoThirdsPcIdAlarms' = False$
$eventname' = CaLessTwoThirdsPcIdAlarms$
$eventdestination' = \{\, Nipc \,\}$
$eventtime' = Time$

Two-thirds power companies have their generators' intrusion detection alarms going on and off:

S — *CaConditionSchemaTwoThirdsPcsWithGensIdAlarms*

$\Delta ControlAreaFSM$
$\Delta OutputEvent$

$(\#\, PcsWithGensIdAlarms = 2) \land (TwoThirdsPcsWithGensIdAlarms = False)$
$TwoThirdsPcsWithGensIdAlarms = True$
$eventname' = CaTwoThirdsPcsWithGensIdAlarms$
$eventdestination' = \{\, Nipc \,\}$
$eventtime' = Time$

S — *CaConditionSchemaLessTwoThirdsPcsWithGensIdAlarms*

$\Delta ControlAreaFSM$
$\Delta OutputEvent$

$(\#\, PcsWithGensIdAlarms < 2) \land (TwoThirdsPcsWithGensIdAlarms = True)$
$TwoThirdsPcsWithGensIdAlarms = False$
$eventname' = CaLessTwoThirdsPcsWithGensIdAlarms$
$eventdestination' = \{\, Nipc \,\}$
$eventtime' = Time$

All power companies have their substations' intrusion detection alarms going on and off:

S — *CaConditionSchemaAllPcsWithSubsIdAlarms*

$\Delta ControlAreaFSM$
$\Delta OutputEvent$

$(\#\, PcsWithSubsIdAlarms = \#\, MyPcs)$
$AllPcsWithSubsIdAlarms = True$
$eventname' = CaAllPcsWithSubsIdAlarms$
$eventdestination' = \{\ Nipc\ \}$
$eventtime' = Time$

S — *CaConditionSchemaNotAllPcsWithSubsIdAlarms*

$\Delta ControlAreaFSM$
$\Delta OutputEvent$

$(\#\, PcsWithSubsIdAlarms \neq \#\, MyPcs) \wedge (AllPcsWithSubsIdAlarms = True)$
$AllPcsWithSubsIdAlarms = False$
$eventname' = CaNotAllPcsWithSubsIdAlarms$
$eventdestination' = \{\ Nipc\ \}$
$eventtime' = Time$

Finite-State Machine File: ControlRegionFSM

Initialization Schemas

S — *InitControlRegionFSM*

$ControlRegionFSM$

$MyCr = 0$
$Nipc = 0$
$MyCas = \varnothing$
$CasUp = \varnothing$
$CasDown = \varnothing$
$NodeDown = False$
$DbDown = False$
$IdAlarm = False$

$$
\begin{array}{|l}
\hline
S \quad \textit{InitCrMyCr} \\
\Delta ControlRegionFSM \\
crid? : \mathbb{N} \\
\hline
MyCr' = crid? \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
S \quad \textit{InitCrNipc} \\
\Delta ControlRegionFSM \\
nipcid? : \mathbb{N} \\
\hline
Nipc' = nipcid? \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
S \quad \textit{InitCrNewCa} \\
\Delta ControlRegionFSM \\
caid? : \mathbb{N} \\
\hline
MyCas' = MyCas \cup \{ caid? \} \\
CasUp' = CasUp \cup \{ caid? \} \\
\hline
\end{array}
$$

Event Schemas (Low-Level/Basic Events)

Local site failure and repair:

$$
\begin{array}{|l}
\hline
S \quad \textit{CrSchemaLocalSiteFailure} \\
\Delta ControlRegionFSM \\
\Delta OutputEvent \\
\hline
NodeDown' = True \\
eventname' = CrSiteFailure \\
eventdestination' = MyCas \\
eventtime' = Time \\
\hline
\end{array}
$$

S *CrSchemaLocalSiteRepair*

$\Delta ControlRegionFSM$
$\Delta OutputEvent$

$NodeDown' = False$
$eventname' = CrSiteRepair$
$eventdestination' = MyCas$
$eventtime' = Time$

Local database failure and repair:

S *CrSchemaLocalDbFailure*

$\Delta ControlRegionFSM$
$\Delta OutputMessage$

$DbDown' = True$
$msgname' = MsgCrLocalBalancing$
$msgdestination' = \{ MyCr \}$
$msgtime' = Time$

S *CrSchemaLocalDbRepair*

$\Delta ControlRegionFSM$
$\Delta OutputMessage$

$DbDown' = False$
$msgname' = MsgCrNormalBalancing$
$msgdestination' = \{ MyCr \}$
$msgtime' = Time$

Local intrusion detection alarm going on and off:

S *CrSchemaLocalIdAlarmOn*

$\Delta ControlRegionFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$

$IdAlarm' = True$
$msgname' = MsgCrRaiseAlert$
$msgdestination' = \{ MyCr \}$
$msgtime' = Time$
$eventname' = CrIdAlarmOn$
$eventdestination' = MyCas \cup \{ Nipc \}$
$eventtime' = Time$

S *CrSchemaLocalIdAlarmOff*

$\Delta ControlRegionFSM$
$\Delta OutputMessage$
$\Delta OutputEvent$

$IdAlarm' = False$
$msgname' = MsgCrLowerAlert$
$msgdestination' = \{ MyCr \}$
$msgtime' = Time$
$eventname' = CrIdAlarmOff$
$eventdestination' = MyCas \cup \{ Nipc \}$
$eventtime' = Time$

Control area site failure and repair:

S *CrSchemaCaSiteFailure*

$\Delta ControlRegionFSM$
$\Delta OutputMessage$
$caid? : \mathbb{N}$

$CasDown' = CasDown \cup \{ caid? \}$
$CasUp' = CasUp \setminus \{ caid? \}$
$msgname' = MsgCrSwitchToBackupCa$
$msgdestination' = \{ MyCr \}$
$msgtime' = Time$

S *CrSchemaCaSiteRepair*

$\Delta ControlRegionFSM$
$\Delta OutputMessage$
$caid? : \mathbb{N}$

$CasUp' = CasUp \cup \{ caid? \}$
$CasDown' = CasDown \setminus \{ caid? \}$
$msgname' = MsgCrSwitchBackFromBackupCa$
$msgdestination' = \{ MyCr \}$
$msgtime' = Time$

Finite-State Machine File: NipcFSM

Initialization Schemas

S *InitNipcFSM*

$NipcFSM$

$MyNipc = 0$
$AllCrs = \varnothing$
$AllCas = \varnothing$
$CrsIdAlarms = \varnothing$
$CasIdAlarms = \varnothing$
$CasPcsIdAlarms = \varnothing$
$CasGensIdAlarms = \varnothing$
$CasSubsIdAlarms = \varnothing$
$CrCoordinatedAttack = False$
$CaCoordinatedAttack = False$
$PcCoordinatedAttack = False$

S *InitNipcMyNipc*

$\Delta NipcFSM$
$nipcid? : \mathbb{N}$

$MyNipc' = nipcid?$

S — *InitNipcNewCa*

$\Delta NipcFSM$
$caid? : \mathbb{N}$

$AllCas' = AllCas \cup \{ caid? \}$

S — *InitNipcNewCr*

$\Delta NipcFSM$
$crid? : \mathbb{N}$

$AllCrs' = AllCrs \cup \{ crid? \}$

Event Schemas (Low-Level/Basic Events)

Control region intrusion detection alarm going on and off:

S — *NipcSchemaCrIdAlarmOn*

$\Delta NipcFSM$
$crid? : \mathbb{N}$

$CrsIdAlarms' = CrsIdAlarms \cup \{ crid? \}$

S — *NipcSchemaCrIdAlarmOff*

$\Delta NipcFSM$
$crid? : \mathbb{N}$

$CrsIdAlarms' = CrsIdAlarms \setminus \{ crid? \}$

Control area intrusion detection alarms going on and off:

S — *NipcSchemaCaIdAlarmOn*

$\Delta NipcFSM$
$caid? : \mathbb{N}$

$CasIdAlarms' = CasIdAlarms \cup \{ caid? \}$

S — *NipcSchemaCaIdAlarmOff*

$\Delta NipcFSM$
$caid? : \mathbb{N}$

$CasIdAlarms' = CasIdAlarms \setminus \{ caid? \}$

Control area power companies' intrusion detection alarms going on and off:

S — *NipcSchemaCaTwoThirdsPcIdAlarms*

$\Delta NipcFSM$
$caid? : \mathbb{N}$

$CasPcsIdAlarms' = CasPcsIdAlarms \cup \{ caid? \}$

S — *NipcSchemaCaLessTwoThirdsPcIdAlarms*

$\Delta NipcFSM$
$caid? : \mathbb{N}$

$CasPcsIdAlarms' = CasPcsIdAlarms \setminus \{ caid? \}$

Control area power companies' generator intrusion detection alarms going on and off:

S — *NipcSchemaCaTwoThirdsPcsWithGensIdAlarms*

$\Delta NipcFSM$
$caid? : \mathbb{N}$

$CasGensIdAlarms' = CasGensIdAlarms \cup \{ caid? \}$

S — *NipcSchemaCaLessTwoThirdsPcsWithGensIdAlarms*

$\Delta NipcFSM$
$caid? : \mathbb{N}$

$CasGensIdAlarms' = CasGensIdAlarms \setminus \{ caid? \}$

Control area power companies' substation intrusion detection alarms going on and off:

S — *NipcSchemaCaAllPcsWithSubsIdAlarms*

$\Delta NipcFSM$
$caid? : \mathbb{N}$

$CasSubsIdAlarms' = CasSubsIdAlarms \cup \{ caid? \}$

S — *NipcSchemaCaNotAllPcsWithSubsIdAlarms*

$\Delta NipcFSM$
$caid? : \mathbb{N}$

$CasSubsIdAlarms' = CasSubsIdAlarms \setminus \{ caid? \}$

Event Schemas (High-level Events)

Control region coordinated security attack on and off:

S — *NipcConditionSchemaCrCoordinatedAttack*

$\Delta NipcFSM$
$\Delta OutputEvent$

$(\# CrsIdAlarms \geq 5) \land (CrCoordinatedAttack = False)$
$CrCoordinatedAttack' = True$
$eventname = NipcCrCoordinatedAttack$
$eventdestination = AllCrs$
$eventtime = Time$

S — *NipcConditionSchemaNotCrCoordinatedAttack*

$\Delta NipcFSM$
$\Delta OutputEvent$

$(\# CrsIdAlarms < 5) \land (CrCoordinatedAttack = True)$
$CrCoordinatedAttack' = False$
$eventname = NipcNotCrCoordinatedAttack$
$eventdestination = AllCrs$
$eventtime = Time$

Control area coordinated security attack on and off:

S — *NipcConditionSchemaCaCoordinatedAttack*

$\Delta NipcFSM$
$\Delta OutputEvent$

$(\# CasIdAlarms \geq 72) \land (CaCoordinatedAttack = False)$
$CaCoordinatedAttack' = True$
$eventname = NipcCaCoordinatedAttack$
$eventdestination = AllCas$
$eventtime = Time$